НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

«До захисту допущено»
Завідувач кафедри
_____ О.І. Ролік
«___»_____2019 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6. 050201 «Системна інженерія»

на тему: «Система управління сховищами даних»

Виконав:
студент IV курсу, групи ІА-51
Прокопенко Микита Ігорович _____

Керівник:
Професор, д.т.н. Ролік О.І.

_____

Рецензент:

_____

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2019 рік

Пояснювальна записка
до дипломного проекту
на тему: «Система управління сховищами даних»

Київ – 2019 рік

АНОТАЦІЯ

Прокопенко М.І. Система управління сховищами даних. КПІ ім. Ігоря Сікорського, Київ, 2019.

Пояснювальна записка дипломного проекту складається з 4 розділів, містить 60 сторінок, 17 рисунків, 9 таблиць, посилання на 14 літературних джерел та 4 конструкторські документи.

Ключові слова: система управління, сховище даних, інтеграція, клієнт-серверна архітектура, база даних.

Бакалаврський проект присвячений розробці системи управління сховищами даних. У роботі вирішено задачі побудови системи в цілому та підсистем завантаження даних, пост-обробки даних. Спроектовано єдину модель даних та уніфікований інтерфейс доступу до них.

Отримані результати можуть бути корисними при необхідності роботи із різними форматами даних.

SUMMARY

Kulish D.S. Automated paper production management system. Igor Sikorsky KPI, Kyiv, 2019.

Explanatory Note of thesis project consists of 6 chapters, contains 64 pages, 24 picture, 9 tables, references to 20 literary sources and 4 design documents.

Keywords: control system, data warehouse, integration, client-server architecture, database.

The Bachelor's project is dedicated to the development of a data warehouse management system. In this paper the problems of constructing the system structure in general and a subsystem of data loading, post-processing of data were solved. A single data model and a unified interface for access to them were designed.

The results can be useful in working with different data formats.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД – база даних;

ІС – інформаційна система;

ООП – об’єктно-орієнтоване програмування;

ПЗ – програмне забезпечення;

СУБД – система управління базами даних;

API – Application Programming Interface – інтерфейс прикладного програмування

JSON – JavaScript Object Notation – нотація об'єкта javascript;

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту;

HTTPS – HyperText Transfer Protocol Secure – безпечний протокол передачі гіпертексту;

REST – Representational State Transfer – передача репрезентативного стану;

SDK – Software Development Kit – набір засобів розробки.

ВСТУП

Інформаційні системи (ІС) є невід'ємною складовою сучасного суспільства. Кожна ІС вирішує специфічний спектр задач в залежності від прикладної галузі, для якої призначена та розроблена ця система. Але загальними для кожної сфери діяльності є збір, збереження, пошук та обробка інформації. Без використання таких систем жодна організація не може повноцінно вести свою діяльність та бути конкурентоспроможною. Така система повинна:

- забезпечувати доступ до інформації без суттєвих затримок;
- ефективно оброблювати інформацію з економічної точки зору;
- надавати повну, достовірну й точну інформацію;
- дозволяти аналізувати зміни критичних показників;
- забезпечувати надійний захист даних від несанкціонованого доступу.

Автоматизація професійної діяльності будь-якого спрямування за допомогою комп'ютеризованих систем потребує зберігання певного обсягу даних. При цьому джерелами цих даних можуть виступати сховища, різні за своєю внутрішньою структурою, які розроблені незалежно один від одного та які мають різні способи маніпулювання даними. Кожне таке джерело даних має власне програмне забезпечення, які часто несумісні один з одним, що призводить до розширення кількості використовуваних програм на підприємстві та вимагає використання все більшого об'єму ресурсних компонентів. Для налагодження бізнес-процесів та підвищення ефективності сучасного підприємства виникає необхідність у взаємодії між сховищами даних різного роду на одному рівні, як із одним джерелом даних.

Освітлена вище проблема не є новою і наукові досліди у цій галузі велись вже у 70-х роках. Але через складність та багатогранність цієї проблеми було спроектовано велику кількість систем, які частково вирішують проблему, оскільки більшість з них спрямовано на роботу із окремими типами сховищ

даних. Тому тема розробки системи управління сховищами даних є актуальною.

Метою даної робота є розробка системи, яка дозволяє інтегрувати дані з джерел, незалежно від того, яку структуру вони мають та з якими форматами працюють, та надає користувачу уніфікований інтерфейс представлення даних та роботи з ними.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

На сьогоднішній день існує велика кількість варіантів представлення інформації, що кожен може знайти собі найбільш зручний. З найбільш популярних маємо:

– бази даних;
– файли (Excel, Word);
– веб-сервіси з власними API для завантаження інформації.

Здавалося б, що такий різновид технологій має лише позитивні сторони, але нажаль це не так. Як це зазвичай і трапляється, спочатку створюється широкий спектр різних варіантів реалізацій для вирішення тієї чи іншої проблеми, а згодом такий широкий різновид перестає бути зручним. Сучасний користувач зацікавлений в спрощенні отримання інформації, у відмові від тисячі програм для кожного окремого формату.

В якості прикладу можна привести будь-яке велике підприємство, особливо промислове. В різних відділах підприємства зручніше вести статистику у своєму форматі. Звичайно, що рано чи пізно потрібно підводити підсумки роботи компанії, та для цього необхідно отримати та опрацювати дані з усіх відділів. І саме тут виникає головна проблема – кожен формат потребує окремого програмного продукту для перегляду та редагування інформації, що є максимально не зручно та дещо затратно з точки зору часу та використання ресурсних компонентів обчислювального обладнання.

Системи, які вирішують подібні проблеми, називають системами інтеграції даних.

1.1 Поняття та призначення системи інтеграції даних

Система інтеграції даних – це сукупність програмних та лінгвістичних компонентів, які в цілому забезпечують єдиний уніфікований інтерфейс для

доступу до певної множини неоднорідних та незалежних джерел даних [1]. Таким чином, інформаційні ресурси з усієї кількості інтегрованих джерел представляється користувачеві у вигляді одного єдиного джерела даних.

Робота із системою інтеграції даних надає можливість користувачу абстрагуватися від того, з яким саме сховищем даних він працює, як воно влаштоване та яким чином забезпечується доступ до нього. Джерелами даних можуть виступати не лише звичні бази даних із різними моделями даних (реляційними, об'єктно-реляційними, ієрархічними, об'єктно-орієнтованими, графовими та мережевими). Цю роль можуть виконувати файли, які містять структуровані дані, а також веб-сервіси, які працюють із даними та мають власні інтерфейси прикладного програмування для здійснення доступу до таких систем. Таким чином, метою системи інтеграції даних є забезпечення взаємодії із різними сховищами даних, які є незалежними один від одного, на основі єдиної моделі даних.

У даному дипломному проекті було розроблено систему управління сховищами даних, яка включає у себе поняття системи інтеграції. Кількість підтримуваних сховищ даних є обмеженою, але структура системи побудовано таким чином, щоб забезпечити можливість динамічно розширювати цей обсяг джерел. Для демонстрації працездатності системи та можливості взаємодії із будь-якими джерелами, у якості сховищ даних були обрані реляційні бази даних, а саме MySQL та PostgreSQL, файли структурованих даних формату CSV, та Google таблиці, що представляє собою веб-сервіс для віртуального збереження даних.

1.2 Задачі інтеграції даних

Розробка системи інтеграції даних вимагає подолання задач, до яких відносять:

– проектування архітектури системи;

– створення єдиної моделі даних;

– визначення типів даних, використовуваних у системі інтеграції даних;

– інтеграція метаданих, що використовуються у конкретному сховищі даних;

– подолання неоднорідності сховищ даних.

Архітектуру системи управління сховищами даних побудовано на основі моделі клієнт-сервер, причому розроблена система виконує функції лише сервера. У якості архітектурного стилю обрано REST, який забезпечує взаємодію компонентів системи у мережі на основі протоколу HTTP та описує правила його використання.

Протокол HTTP був розроблений для обміну даними використовуючи гіпер-текст формату HTML, але підтримує і використання інших форматів. Єдина модель даних у системи управління сховищами даних використовує формат JSON. Цей формат набув широкої популярності у останнє десятиріччя і став неофіційним стандартом у архітектурному стилі REST. Це зумовлено кількома причинами. По-перше, більшість клієнтських додатків, осболиво у Web, використовують мову програмування JavaScript, а об’єкти формату JSON вже є об’єктами, якими оперує дана мова, тобто відпадає необхідність у перетворені об’єктів з одного виду до іншого. По-друге, даний формат є компактним та інтуїтивно зрозумілим.

У якості типів даних було обрано найбільш популярні, які підтримуються більшістю сховищами даних:

– цілочисельні;

– десяткові;

– текстові;

– дати.

Неоднорідність сховищ даних подолано виділенням єдиного інтерфейсу взаємодії із адаптерами, які забезпечують взаємодію усієї системи із конкретними джерелами даних.

1.3 Зв’язок висвітленої проблеми із важливими науковими та практичними завданнями

Зі збільшенням обсягу даних, що використовуються, зростає і роль інтеграції даних. Через це даній темі присвячено широкий склад теоретичних трудів. Багато аспектів даної галузі залишаються невирішеними, оскільки проблема інтеграції даних надзвичайно багатоаспектна. Від задач, які постають у процесі інтеграції даних, залежить складність використовуваних підходів до вирішення даних проблем.

1.4 Проблеми інтеграції даних

При розробці системи управління сховищами даних виявлено наступні проблеми інтеграції даних:

- різнорідність сховищ даних;
- автономність сховищ даних;
- розподіленість сховищ даних.

Під різнорідністю розуміється те, що кожне сховище даних розроблювалося окремо та має не тільки різні моделі даних, але й різний формат метаданих.

Кожне сховище даних екстплуатується незалежно один від одного, тобто є автономним.

Також важливою проблемою є те, що джерела фізично або логічно розподіленість у мережі. Доступ до кожного з них здійснюється на основі мережевих протоколів.

Також при проектуванні систем інтеграції даних часто виділяють наступні конфлікти схем даних:

– неоднорідність моделей;

– конфлікти іменування;

– структурні конфлікти (одні і ті самі сутності подають у різних джерелах різними структурами даних).

Описані вище проблеми подолано наступним чином: для кожного сховища даних розроблюється власний адаптер, який використовує будь-які підходи до взаємодії із джерелом даних, які визначаються його інтерфейсом прикладного програмування. Таким чином, робота із конкретним сховищем даних виконується автономно та не впливає на роботу усієї системи.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБГРУНТУВАННЯ ПРИНЦИПІВ ПОБУДОВИ СИСТЕМИ

2.1 Аналіз останніх досліджень та публікацій

Під інтеграцією даних в ІС мається на увазі гарантування уніфікованого інтерфейсу з метою надання доступу до певних даних, які у свою чергу отримано з різнорідних незалежних сховищ. Отже, для користувача інформаційні ресурси всієї сукупності інтегрованих джерел утворюють єдиний сервіс.

Проблемою динамічної інтеграції інформаційних ресурсів наукові та інженерно-технічні спільноти світу переймаються вже багато років, але тільки в останні роки поява і активне розвинення сучасних інфраструктур (WEB та GRID середовища) і відкритих сервіс-орієнтованих архітектур у галузі інформаційних технологій (SOA, OGSA), а також значні просування в розробці відповідних міжнародних базових стандартів обміну інформацією (XML, RDF, TM, OWL) дозволяють створювати принципово нові моделі ІС[]. Вони надають можливість будувати глобально розподілені застосунки, що реалізують технологічну послідовність ланок, у якій можуть бути використані як власні ІР, так і запропоновані окремими організаційними структурами. Беруть до уваги також те, що під час взаємодії з подібним застосунком, існує можливість замість одного інформаційного сервісу використати інший або ж видалити, якщо він вже не є актуальним та цінним для роботи, а також з легкістю додати новий.

2.2 Рівні інтеграції даних

Інтеграція даних забезпечується на наступних трьох рівнях [] (рисунок 2.1):

– на фізичному рівні;

– на логічному рівні;
– на глобальному рівні.

Рисунок 2.1 – Рівні інтеграції даних

Теоретично на фізичному рівні інтеграція даних — це не складне завдання, адже полягає в перетворенні даних з різноформатних джерел у єдиний стандарт їх фізичного подання.

На логічному рівні інтеграція даних полягає в забезпеченні доступу до даних, розташованих у джерелах в межах однієї глобальної схеми, яка містить опис їх спільного представлення, зважаючи на структурні властивості даних та іноді поведінкові (якщо йде мова про моделі об’єктів). На цьому рівні семантичні властивості не беруть до уваги.

На глобальному рівні інтеграція даних передбачає підтримку уніфікованого подання даних, але не тільки з урахуванням структурних та поведінкових властивостей, а й семантичних у зв’язку з цілісністю онтології предметної області .

На кожному з рівнів існують певні проблеми для інтеграції даних у зв’язку з неоднорідністю джерел. А саме:

– на фізичному рівні інтеграції даних можуть перешкоджати різні формати файлів, що містяться в джерелах;

– для логічного рівня перешкодою може стати відмінність схем даних в межах однієї моделі або використання різних моделей для кожного з джерел. Наприклад, деякі з джерел — це web-сайти, а деякі бази даних тощо;

– на верхньому, глобальному, рівні може виникнути ситуація, коли для кожного з джерел поставлена у відповідність окрема онтологія. Наприклад, коли певна частина предметної області з її унікальними поняттями змодельована окремо. До того ж кожній такій моделі відповідає окреме джерело, але у дійсності дані частини предметної області перетинаються.

Джерела даних мають різнорідні властивості та характеристики, які здійснюють великий вплив на визначення методу інтеграції. А саме вони можуть бути динамічними, статичними, однорідними або ж неоднорідними (відповідно до того, про який рівень інтеграції йде мова).

2.3 Способи інтеграції даних

2.3.1 Фізична інтеграція

Існують два фундаментальних підходи до вирішення поставленої проблеми. Перший підхід пов'язаний з побудовою сховищ даних, коли інтегровані дані з різних джерел трансформуються відповідно до цільової моделі і розміщаються в локальні, або підключені до серверів, носії даних. Найпростішим із способів збереження інформації є Direct Attached Storage (DAS). Він є найпростішим, очевидним та інтуїтивно зрозумілим.

Головними і незаперечними перевагами такого підходу є легкість адміністрування зібраних на одному сервері даних, повний контроль наданих даних, уніфікація даних, що зберігаються вже на фізичному рівні.

Однак на цьому всі переваги закінчуються. Сховища даних будуть жити і розвиватися в подальшому, але їх застосування все більше буде обмежуватися рішенням задач зберігання і доступу до даних в рамках одної, досить унітарної організації.

На сьогоднішній день бурхливий розвиток мережі призвів до того, що єдиним розумним способом впоратися з наростаючим інформаційним потоком є організація мережевого способу обробки і зберігання даних. Тому наступним еволюційним кроком у розвитку поняття сховища даних стали системи класу NAS (Network Attached Storage) – мережеві системи зберігання даних. Ці системи використовують стандартні IP-мережі і мережеві файлові системи. Саме NAS-системи прийшли на зміну DAS-структурам, що підключається до мережі, при організації загальнодоступних багатотомних архівів.

Підхід NAS включає в себе файлову систему, використовувану на базі відповідної мережевий операційної системи, яка може налаштовуватися, і файл-сервер, підключений до мережі. На відміну від DAS, технологія NAS підтримує доступ на рівні файлів, а не блоків даних. Наслідок цієї відмінності – методика спільного використання файлів різними мережевими додатками. Саме NAS бере на себе задачу трансляції звернення до конкретного файлу на запит на рівні блоків даних і екранує мережеві призначені для користувача процеси від проблем, пов'язаних із забезпеченням актуальності.

Технологія NAS більш ефективна в порівнянні з DAS, проте в рамках мережевої концепції обробки даних вона все ж не може бути визнана рішенням, хоча б тому, що з кількох NAS-пристроїв неможна створити єдиний пул ресурсів зберігання.

Можна сказати, що найбільш технологічно розвиненим рішенням організації сховища даних сьогодні є технологія SAN (Storage Area Networking), що надає для мережевої сукупності серверів консолідований мережевий ресурс зовнішньої пам'яті без навантаження на локальну мережу. Консолідація в

ідеології SAN будується на основі створення загального пулу ресурсів, що представляють мережеву структуру з єдиними принципами доступу.

Ідеологія SAN має на увазі під собою фізичну однаковість ресурсів. Крім того, для реалізації вона вимагає рішень на основі єдиних технічних засобів, що, не дивлячись на всі її плюси, робить її маловживаною для доступу до розподілених по мережі гетерогенних інформаційних ресурсів. Найчастіше проблема інтеграції гетерогенних даних формується в такий спосіб: є кілька гетерогенних джерел даних, які якимось чином пов'язані на смисловому рівні; необхідно надати можливість уніфікованого доступу до цих даних, як якщо б вони мали єдине логічне і фізичне представлення.

2.3.2 Віртуальна інтеграція

Другий підхід пов'язаний з поняттям віртуальної інтеграції гетерогенних джерел даних, коли дані не матеріалізуються в локальній базі даних, а використовується проміжне програмне забезпечення, яке транслює запити користувачів в специфічні запити до джерел і формує остаточний результат.

Підхід цих систем характеризується насамперед тим, що інтегруються дані з чіткою структурою (хоча структура даних різних джерел може змінюватися). На наступному етапі з'явилися системи інтеграції, засновані на використанні медіаторів і створювані на базі напівструктурованих даних. Виникнення XML і супутніх технологій (XSLT, Xquery) викликало сплеск нових розробок технології віртуальної інтеграції. Загальний принцип функціонування систем, які підтримують віртуальну інтеграцію даних, зображено на рисунку 2.2.

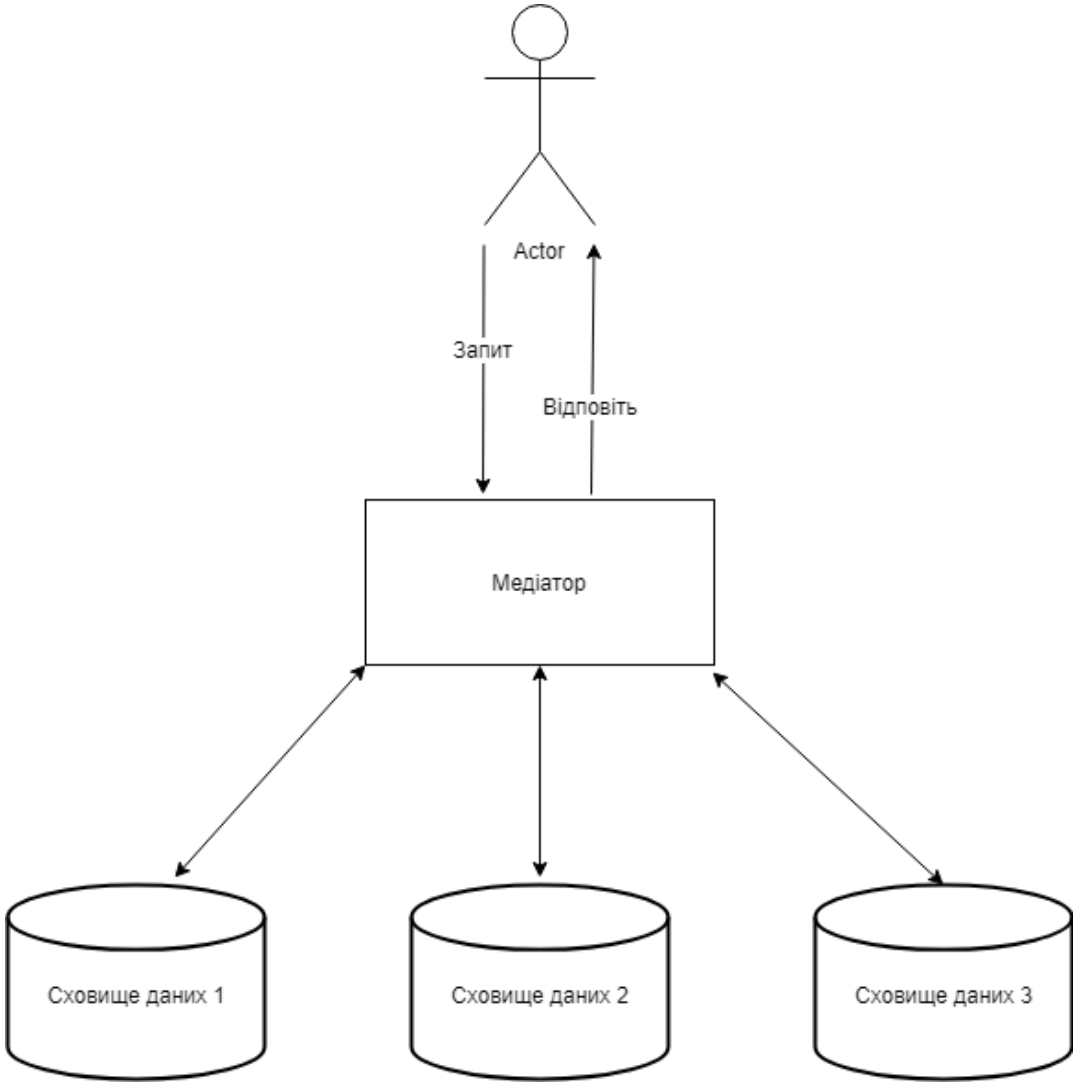

Рисунок 2.2 – Принцип функціонування системи з підтримкою віртуальної інтеграції даних

Віртуальний підхід у більшості випадків є актуальним при застосуванні джерел даних, які часто оновлюються. Результатом другого підходу на етапі інтеграції є сформоване матеріалізоване представлення даних, яке значно відрізняється від першоджерел.

2.4 Архітектура систем інтеграції

Серед архітектур, які використовуються в системах інтеграції даних, найпоширенішою стала архітектура з посередником. Для посередника

основними задачами є: підтримка єдиного призначеного для користувача інтерфейсу, що засновується на глобальному поданні даних, які знаходяться в джерелах, та підтримка відображення між глобальним представленням та локальним. Над запитом користувача, який сформовано в межах уніфікованого інтерфейсу, виконується декомпозиція, тобто розбиття на деяку кількість підзапитів. Кожен з підзапитів адресується необхідному локальному джерелу даних. Базуючись на результатах їх обробки, формується цілісна відповідь на користувацький запит.

Популярності набули два різновиди архітектури з посередником: Global as View та Local as View [1, 2].

Різновид Global as View окреслює глобальне представлення даних, які інтегруються, в поняттях певних представлень локальних джерел. Даний різновид результативний у випадку, коли визначена множина джерел, що використовуються. У разі, коли основною метою системи інтеграції є підтримка цілісного матеріалізованого представлення даних, що інтегруються, то їх перетворення в уніфіковане глобальне представлення виконується один раз.

Якщо застосовується різновид Local as View, то представлення усіх даних локальних джерел окреслюються поняттями певного глобального інтегруючого подання. У даному випадку стає складнішим відображення користувацьких запитів в середовище локальних джерел. Проте даний підхід дає можливість динамічно змінювати їх склад. Підключати нове локальне джерело даних можна як на етапі розробки, так і роботи.

2.5 Процес вилучення інформації з різноструктурованих даних та приведення їх до цільової схеми

Основні етапи процесу вилучення сутностей із вхідних різноструктурованих сховищ даних, їх інтеграції для подальшого аналізу отриманої інформації при вирішені прикладної задачі, наведено на рисунку 2.3.

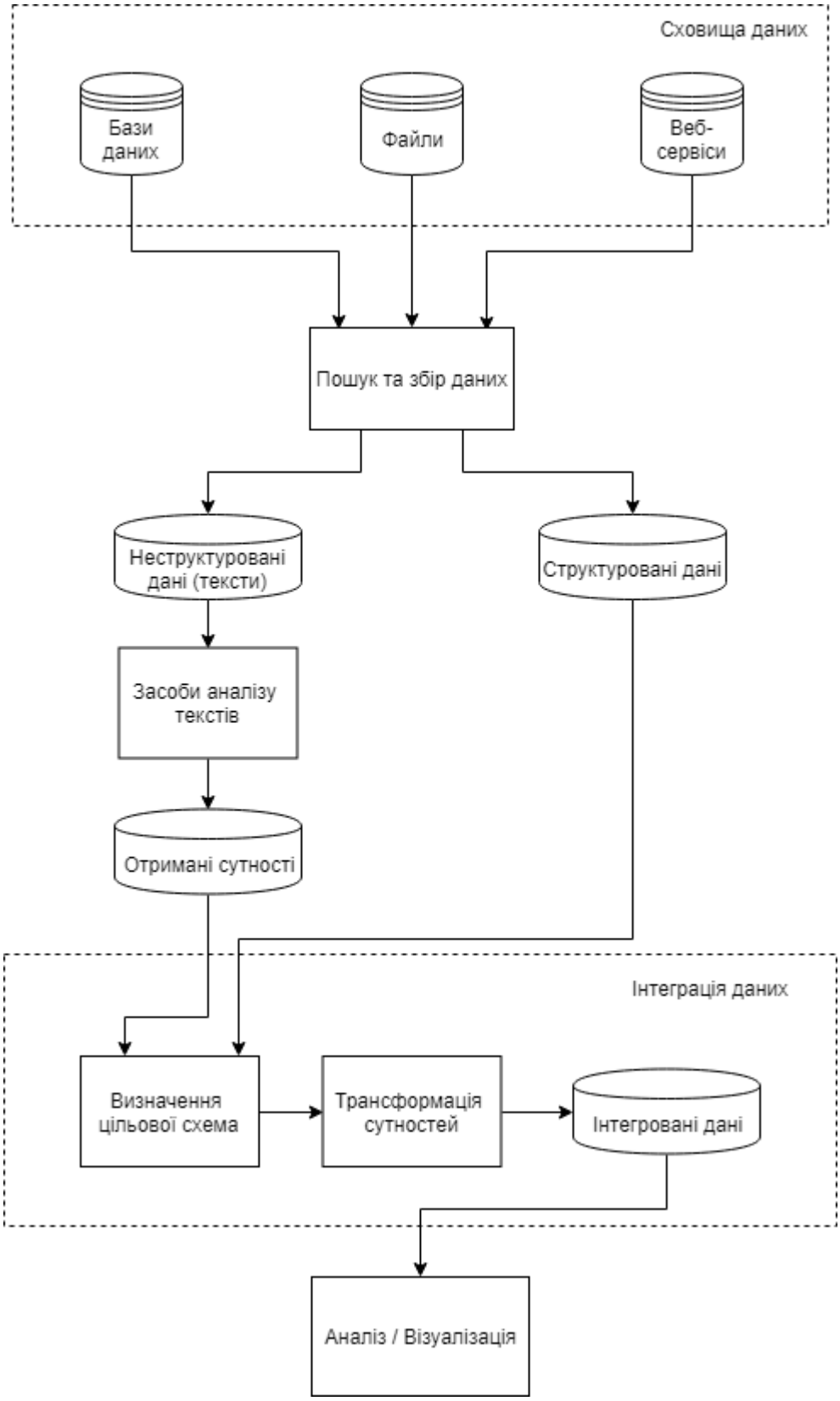

Рисунок 2.3 – Етапи процесу збору різноструктурованих даних та їх інтеграції

Процес починається з пошуку інформаційних ресурсів, які відповідають задачі, та вилучення з них вхідних колекцій даних. Інформаційні ресурси можуть містити структуровані дані (бази даних, структуровані файлові формати даних), слабоструктуровані дані (наприклад, дані з соціальних мереж) та неструктуровані дані (тексти).

Протягом наступного етапу неструктуровані дані пропускаються через засоби аналізу текстів [4], такі як Pillenti, AQL, System. При цьому з текстів вилучаються сутності, наприклад, персони, організації, територіальні виникнення та інше. Сутності, які отримуються з тестів одним конкретним інструментальним засобом, завжди відповідають однаковій структурі, яка визначається вихідним форматом засобів аналізу текстів. Така структура називається вихідною схемою даних.

2.6 Інтегруючі моделі даних

У більшості випадків як глобальну модель даних для підтримки уніфікованого користувацького інтерфейсу використовують прості поширені моделі, такі як об’єктна, ієрархічна або ж реляційна. Але останнім часом відбувається розширення розробок web-додатків, тому все більшої популярності починає набувати у даному амплуа інтегруюча модель даних, яка базується на стандартах XML.

При застосуванні для окремих сховищ даних різних моделей іноді виникає необхідність у створенні специфічної інтеграційної моделі даних з метою підтримки подання на глобальному рівні. Експериментальні розробки таких моделей почали проводитися ще з середини 70-х років і ведуться до теперішнього часу []. Варто звернути увагу на такого типу моделі, які створені досить нещодавно, адже деякі з них можуть забезпечити подання як повністю структурованих даних, так і слабо структурованих. Досить потужною (у функціональному аспекті) стала модель реалізована мовою «Синтез».

У процесі створення інтегруючих моделей даних іноді застосовується метод, який базується на інтеграції моделей, що підтримуються різними джерелами. Це забезпечує вирішення задачі двоїстості — підтримку сукупності різнорідних представлень для одних даних. За допомогою цього методу подібну модель було створено в кінці 70-х років. Яскравим прикладом є інтеграція реляційної моделі та мережевої (CODASYL).

Наприкінці 80-х років з'явилися перші розробки комбінованої моделі, об'єктно-реляційної. У зв'язку з цим до стандарту мови SQL 1999 року було додано об'єктне розширення (для серверів баз даних).

До вищезазначеного виду засобів інтеграції даних варто віднести і компонент з назвою SQL/XML. Це розширення SQL входить до стандарту 200n. Можливості компоненту передбачають представлення схем баз даних SQL та реляційних даних у форматі XML-файлів.

Нова технологічна платформа Web, заснована на стандартах XML, в останні роки привертає увагу багатьох фахівців як ефективний інструмент інтеграції інформаційних ресурсів у більшостіпрактично важливих випадках [4, 5]. Особлива увага до формату XML викликана не лише його особливостями як описової мови для даних, а й можливістю застосування для передачі повідомлень в мережі Інтернет.

Поглиблена зацікавленість до інтеграції web-ресурсів та реляційних баз даних проявляється до сьогодні. Яскравим прикладом є стандарт XQuery платформи XML. Він виконує функції подібні інтегруючій моделі даних.

2.7 Огляд існуючих засобів інтеграції даних

2.7.1 BizQuery

BizQuery — це пакет серверів та інструментів для розробки додатків, які використовують різнорідні джерела даних. Головним компонентом пакету є сервер інтеграції BizQuery Integration Server, який призначений для здійснення

запитів на мові XQuery. Він підтримує концепцію глобальної схеми даних, яка визначена в XML. Глобальна схема створюється для представлення області конкретного додатка, і на неї відображаються джерела даних. BizQuery підтримує віртуальний підхід: користувач робить запит над глобальною схемою, система інтеграції даних переформулює його в запит над джерелами даних і виконує.

Основні технічні особливості BizQuery наступні:

– підтримка відкритих стандартів гарантує легкість програмування та інтеграції з існуючими додатками (XML, простору імен XML, XSLT, XQuery, OMG XMI, OMG UML і т. д.);

– дворівневий інтерфейс надає два способи здійснення запиту даних: в термінах XML або в термінах UML-моделі;

– підтримка XQuery зручна для користувачів завдяки виразній потужності цієї мови;

– автоматичне створення (при чому чке можна і налаштувати) трьох видів призначеного для користувача інтерфейсу запитів (форми, графічні карти для запитів з підтримкою UML-нотації, каталоги);

– легка взаємодія з будь-якими мовами і середовищами з використанням протоколу SOAP, відкритий інтерфейс web-служб надає можливість інтеграції для додатків, написаних на будь-якій мові програмування (Java, VB, Perl, C / C++, C #, Python).

Але ця система не вирішує проблем, освітлених в даній роботі, оскільки вона підтримує лише ті сховища даних, які працюють із XML форматом.

2.7.2 PrestoDB

PrestoDB - високопродуктивний розподілений механізм запитів SQL для великих даних (Big Data). Система є високо розподіленим механізмом запитів, який побудований для ефективної аналітики з низькою затримкою. Система

підтримує різноманітні випадки використання: спеціальні аналітичні засоби при інтерактивній швидкості, масові багаточасові пакетні запити. Переваги:

– висока ефективність;
– універсальність;
– підтримує Hadoop, S3, Cassandra, MySQL та інші сховища даних;
– працює з інструментами BI.

Але разом із цим, широкий спектр функціональності призводить до ускладнення роботи із даними та вимагає певної технічної підготовки користувачів.

2.8 Обґрунтування принципів побудови системи управління сховищами даних

Проаналізувавши теоретичні відомості у області інтеграції даних та розглянувши існуючі програмні рішення, було сформовано наступні вимоги до системи управління сховищами даних:

– забезпечення єдиної моделі даних;
– побудова архітектури системи на основі моделі «клієнт-сервер»;
– сервіс-орієнтованість сховищ даних;
– можливість динамічно розширювати підтрімувані сховища даних;
– автономність адаптерів сховищ даних.

Єдина модель даних забезпечується за допомогою формату JSON, який був обраний у зв'язку з наступними перевагами:

– зменшення об’єму даних;
– зменшення навантаження як на клієнтську, так і на серверну частину;
– майже необмежені можливості розширення;
– наявність бібліотек для JSON в багатьох мовах програмування;
– підтримка JSON багатьма браузерами.

Особливістю розробленого продукту є відсутність користувацького інтерфейсу. Проект буде реалізовано на основі архітектурного стилю REST, де компоненти системи взаємодіють між собою на основі протоколу HTTP та втілюють модель «клієнт-сервер». Цей варіант було обрано, як найбільш універсальний й відповідний до системи управління сховищами даних.

3 РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ СХОВИЩАМИ ДАНИХ

3.1 Розробка архітектури системи управління сховищами даних

Проаналізувавши існуючі теоретичні відомості та методи вирішення проблеми інтеграції даних різного походження, а також ураховуючи сучасні тенденції у побудові програмного забезпечення, розроблювана система реалізовує наступні загально-архітектурні принципи:

– клієнт-серверна архітектура;

– модульність системи;

– сервіс-орієнтованість сховищ даних;

– підтримка віртуальної інтеграції даних.

Зазвичай, програмні та апаратні компоненти, які входять до складу ІС , не є рівноправними. Це пов’язано з тим, що деякі з них володіють певними важливими ресурсами, а інші мають змогу лише користуватися ними за допомогою надсилання запитів. Тому поставлені перед ІС задачі, умовно поділяють між двома підсистемами – сервером та клієнтом. ІС з архітектурою «клієнт-сервер» дозволяють уникнути ряду проблем, присутніх, наприклад, у файл-серверних системах.

Сервер – центральна частина системи. Саме він забезпечує виконання основного об’єму маніпуляцій з даними, таких як сортування, пошук, агрегація тощо.

Клієнт – складова ІС, яка формує запити до серверу.

На сьогоднішній день такий тип архітектури набув значного поширення та використовується у системах різного масштабу.

Перевагами архітектури «клієнт-сервер» є:

− усі дані, які зберігаються на сервері надійно захищені (забезпечена безпека значно вища, ніж для клієнтів);

− доступ до конкретних даних мають лише клієнти з відповідними правами;

− відсутність дублювання програмами-клієнтами коду програм-серверів;

− занижені вимоги до апаратної частини клієнтів, оскільки усі важливі операції з даними виконує сервер;

− можливість об’єднання різних видів клієнтів, тобто користуватися ресурсами одного сервера можуть клієнти з різними операційними системами, апаратними платформами тощо;

− можливість розвантаження мережі у зв’язку з передачею невеликих порцій даних.

HTTP протокол надає можливість взаємодії компонентів клієнт-серверної архітектури на основі запитів та відповідей. Цей підхід став найбільш розповсюдженим та використовуваним для побудови розподілених систем. HTTP протокол підтримає різні методи, архітектурний стиль REST описує використання наступних чотирьох:

− GET;

− POST;

− PUT;

− DELETE.

Метод GET ініціює отримання даних, метод POST – створення нових даних, метод PUT – оновлення даних та метод DELETE – їхнє видалення з серверу.

3.2 Загальна структура процесу інтеграції даних

В залежності від сформованих вимог до розроблюваної системи, остання має наступні три основні функціональні етапи процесу переносу даних:

– вилучення даних;

– перетворення даних;

– формування вихідних даних та їх передача.

Загальна структура процесу інтеграції даних представлена на рисунку 3.1.

На першому етапі дані витягуються з обраного підтримуваного сховища і готуються до перетворення. Слід зазначити, що для коректного представлення даних після їх завантаження зі сховища у систему повинні вилучатись не тільки самі дані, але й інформація, що описує їх структуру, з якої будуть сформовані метадані для подальшого аналізу даних.

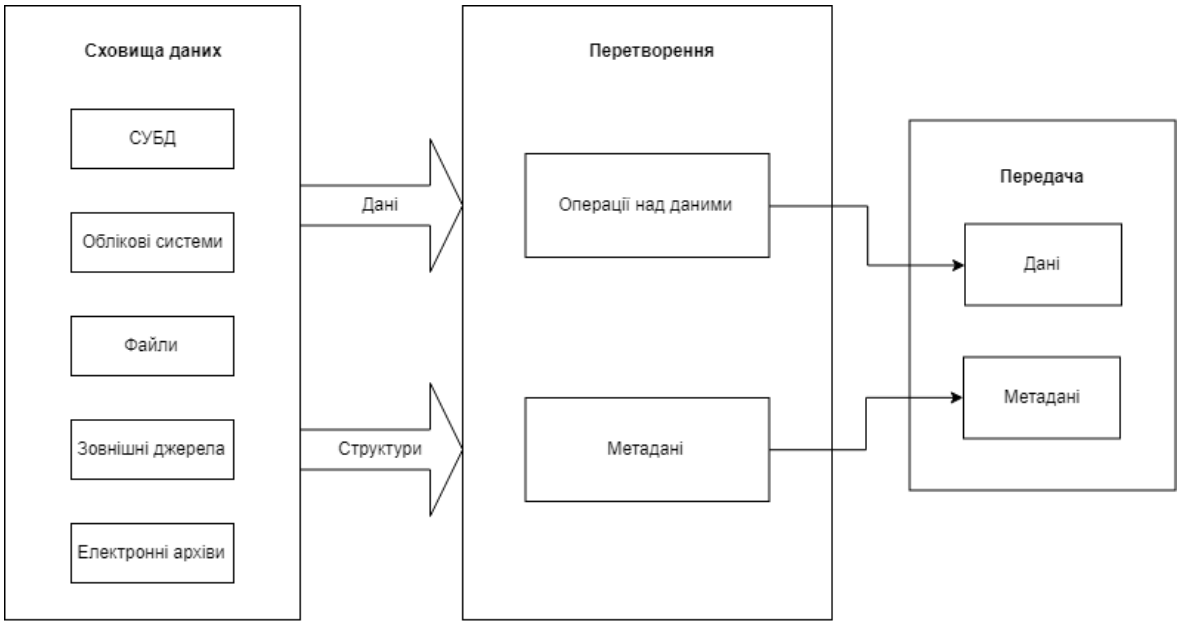

Рисунок 3.1 – Загальна структура процесу інтеграції даних

Наступний етап – перетворення даних. На цьому етапі застосовуються призначені для користувача запити по модифікації інтегрованих даних, наприклад доповнення новими даними на основі вже існуючих (з використанням формул), агрегація даних, їх обмеження, перетворення самих значень а також типів даних.

Останній етап – формування відповіді клієнту на основі перетворених даних.

3.3 Діаграма використання системи управління сховищами даних

Діаграму використання системи управління сховищами даних наведено у документі ІА51.220БАК.005 Д3.

Очевидно, що основною функціональною можливістю системи управління сховищами даних є вилучення даних з обраного джерела. Але через складність цього процесу, що зумовлений різними підходами у взаємодії між системою та сховищами даних, виникає необхідність у підтримці більшої кількості варіантів використання.

Система управління сховищами даних працює із джерелами, як із сервісами. Тобто, доступ до кожного з них є віддаленим, окрім випадків, коли сховищем даних є локальні файли. Таким чином, у систему було додано можливість перевірки з'єднання із обраним сховищем даних.

Усі сховища даних зберігають інформацію у певних об'єктах. Наприклад, для реляційних БД – це таблиці, для локальної файлової системи – це власне файли. Для того, щоб вилучити дані з певного об'єкта, разом із інформацією для підключення до сховища даних, у запиті треба передати унікальний ідентифікатор об'єкта, який буде розпізнаний системою. Для цього реалізовано механізм отримання списку або ієрархії об'єктів з обраного сховища даних.

Кожна операція у роботи зі сховищами даних потребує передачі параметрів для підключення до джерела. Для того, щоб звільнити клієнтів від зберігання великої кількості інформації, цей функціонал перекладено на систему управління сховищами даних. Таким чином, клієнту залишається передати лише ідентифікатор профілю з'єднання, і система автоматично використає дані, збережені у базі.

3.4 Структурна схема системи інтеграції даних

Структурна схема системи управління сховищами даних наведена у документі ІА51.220БАК.005 Д1.

Систему управління сховищами даних розроблено на основі архітектурного стилю REST, який представляє собою набір встановлених правил, що визначають взаємодію компонентів розподіленої системи на основі протоколу HTTP. На рисунку 3.2 зображено фрагмент структурної схеми, який демонструє зв’язок між клієнтськими додатками та системою управління сховищами даних.

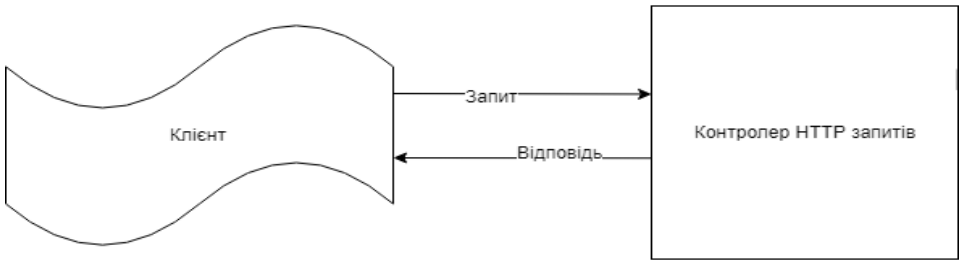

Рисунок 3.2 – Фрагмент структурної схеми

Контролер – це компонент системи, який приймає та оброблює клієнтські HTTP запити та формує відповіді. Поняття контролера у сфері веб-технологій випливає із архітектурного шаблону проектування «Модель–Представлення–Контролер». Даний підхід дозволяє розділити логіку програми на три окремі компоненти:

- модель – втілює «бізнес-логіку» програми, що із даними;
- представлення – відповідає за відображення даних;
- контролер – реагує на дії користувача та ініціює зміну моделі;

Зазвичай під поняттям «представлення» розуміється графічний користувальницький інтерфейс, але у загальному понятті – це спосіб

відображення даних. У системі управління сховищами даних – це формат JSON. Модель надає дані та методи роботи із ними, та не залежить від представлення. У розробленій системі саме компонент «модель» відповідає за роботу із сховищами даних. Контролер – це компонент, що пов’язує між собою модель та представлення.

Зазначений вище підхід реалізовано за допомогою фреймворку Spring Web MVC. Це популярна платформа, призначена для розробки веб-додатків мовою програмування Java.

Логіка роботи Spring MVC побудована навколо класу DispatcherServlet, який приймає та оброблює HTTP запити та відповіді на них. Робочий процес обробки запиту зображено на рисунку 3.3:

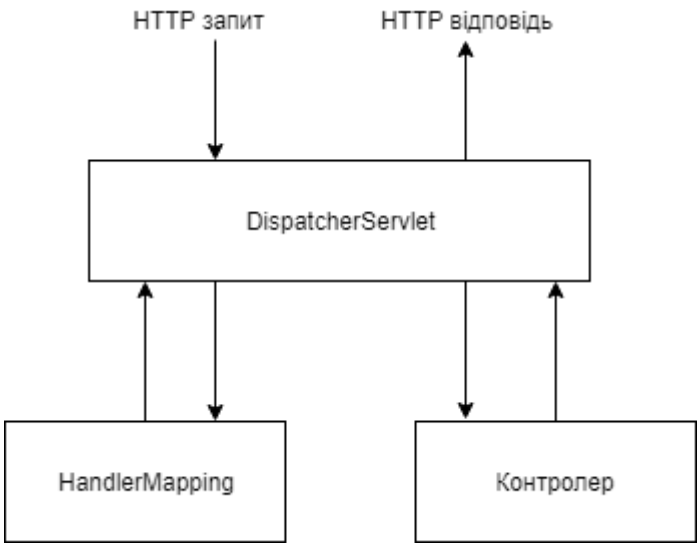

Рисунок 3.3 – Процес обробки запиту у Spring MVC

Після отримання HTTP запиту DispatcherServlet звертається до інтерфейсу HandlerMapping, який визначає, який контролер повинен бути викликаний, після чого відправляє запит в потрібний контролер.

Контролер приймає запит і викликає відповідний службовий метод, заснований на HTTP методі GET або POST. Викликаний метод ініціює зміну моделі у відповідності до клієнтського запиту. Після цього контролер повертає

певний об’єкт, який містить отриману інформацію, або формує виключення, щоб сповістити клієнта про виникнення помилки при виконанні запиту. Далі ця інформація передається об’єкту DispatcherServlet, який вже формує остаточну HTTP відповідь.

DispatcherServlet відправляє запит контролерам для виконання певних функцій. Анотація Controller вказує, що конкретний клас є контролером. Анотація RequestMapping використовується для зв'язування URL із класом або з конкретним методом обробника. Приклад програмного коду контролера у Spring MVC наведено на рисунку 3.4:

```
@Controller
@RequestMapping("/job")
public class BaseController {
    @RequestMapping(method = RequestMethod.POST)
    public String processRequest(Request request) {
        ...
    }
}
```

Рисунок 3.4 – Приклад програмного коду контролера у Spring MVC

У розробленій системі управління сховищами даних складову шаблону MVC «модель» розбито на два модулі:

- модуль завантаження даних;
- модуль пост-обробки даних.

Таке розбиття зумовлено тим, що з точки зору системи інтеграції даних, розроблена система підтримує віртуальну інтеграцію, тобто відповідь на запит формується «на льоту». Таким чином, при кожному запиті відбувається взаємодія із конкретним сховищем даних, з якого завантажується інформація та приводиться до єдиного стандартизованого формату. Другий етап – це застосування функцій перетворення даних на основі запиту, тобто виконується пост-обробка вже після завантаження даних з обраного джерела.

Структуру модулю завантаження даних наведено на рисунку 3.5:

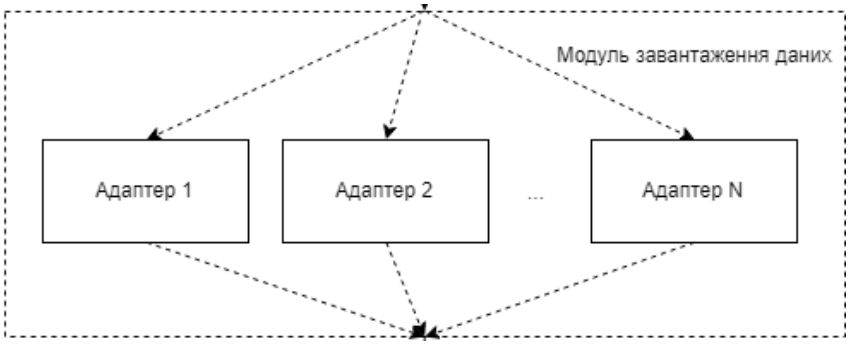

Рисунок 3.5 – Структура модулю завантаження даних

Модуль завантаження даних складається з набору адаптерів. Адаптер – це підмодуль, який працює із відповідним сховищем даних та інкапсулює методи доступу до них. Кожен адаптер надає загальний уніфікований інтерфейс, завдяки чому досягнута можливість динамічно розширювати склад підтримуваних сховищ даних у системі без внесення змін до окремих компонентів.

Також, завдяки даному підходу, контролер виконує функції диспетчера. На основі запиту контролер ініціалізує об’єкт адаптера та викликає відповідний метод.

Після завершення процесу завантаження даних зі сховища, сформована відповідь потрапляє до модулю пост-обробки даних, структуру якого наведено нижче на рисунку 3.6:

На даному етапі до завантажених даних застосовуються користувацькі запити щодо перетворення вихідних даних за допомогою функціоналу, що розроблено у системі управління сховищами даних, а саме:

- фільтрація даних;
- агрегація даних;
- сортування та обмеження;
- генерація нових даних;
- перевизначення типів.

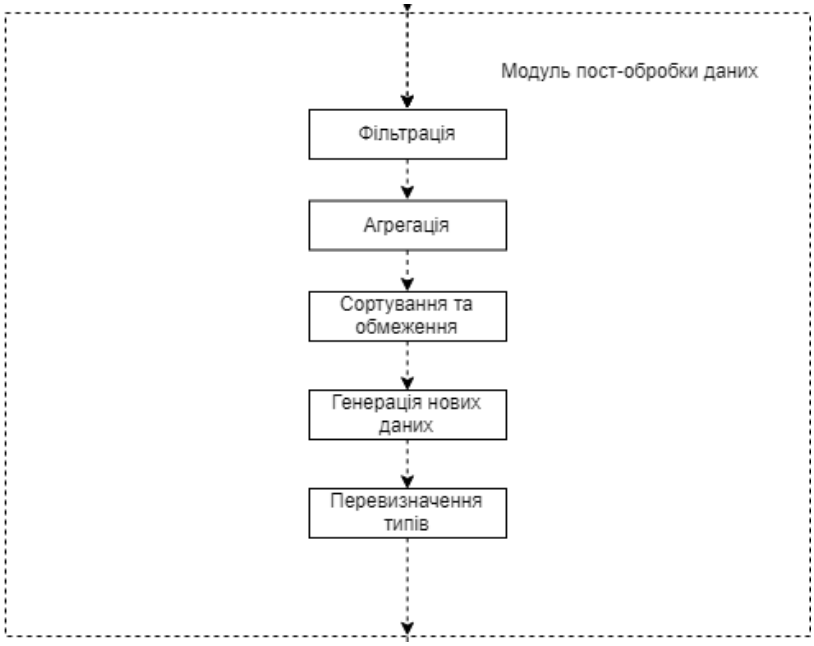

Рисунок 3.6 – Структура модулю пост-обробки даних

Кожне з перелічених перетворень відбувається незалежно один від одного і не є обов’язковим, та реалізовано у окремих підмодулях, що буде розглянуто далі.

Останній етап – отримані дані потрапляють до контролера, який формує остаточну відповідь та передає її клієнту.

3.5 Розробка модулю завантаження даних

Кожне сховище даних, з яким працює система, має свою унікальну архітектуру і взаємодія з ними реалізується різноманітними способами. Але загальний підхід для цього завжди один – використання інтерфейсу прикладного програмування (API) або набору засобів розробки (SDK).

У комп’ютерному програмуванні інтерфейсом прикладного програмування називають набір програмних інструментів (класів, функцій), підпрограм, протоколів зв’язку та інших засобів для побудови програмного забезпечення. API часто використовується як засіб для інтеграції програмних

додатків. Інтерфейс прикладного програмування дає можливість використовувати функціональність програми, та при цьому дозволяє абстрагуватися від того, як саме вона реалізована.

Нижче наведено таблицю 3.1, яка демонструє відповідність між сховищами даних, які розглядаються у даній роботі, та програмними способами взаємодії цих сховищ із системою.

Таблиця 3.1 – Сховища даних та способи доступу

Сховище даних	Спосіб доступу
Реляційні бази даних (MySQL, PostgreSQL та інші)	JDBC
CSV файли	Інструменти стандартного пакету java.io
Google Таблиці	Google Sheets API

В основі JDBC лежить концепція так званих драйверів, що дозволяють отримувати з'єднання з базою даних по спеціально описаному URL. Драйвери можуть завантажуватись динамічно (під час роботи програми). Завантажившись, драйвер сам реєструє себе й викликається автоматично, коли програма вимагає URL, що містить протокол, за який драйвер відповідає.

Перевагами JDBC вважається:

– легкість розробки: розробник може не знати специфіки бази даних, з якою працює;

– код не змінюється, якщо компанія переходить на іншу базу даних;

– не потребує встановлення громіздкої клієнтської програми;

– до будь-якої бази можна підключитись через легко описуваний URL.

Доступ до даних CSV файлів здійснюється за допомогою стандартного пакету java.io, який надає інтерфейс прикладного програмування для введення і виведення інформації через потоки даних, серіалізацію та файлову систему [6].

Для взаємодії із електронними таблицями Google існує Google Sheets API. Розробники цього продукту також надають набір бібліотек для різних мов програмування, в тому числі і для Java. Основні можливості, які надає Google Sheets API:

– імпорт, експорт і форматування даних аркушів;
– керування умовним форматуванням;
– створення та редагування діаграм, вбудованих в аркуш;
– налаштування перевірки даних;
– додавання та коригування фільтрованого перегляду.

Отже, взаємодія із різними сховищами даних потребує різних підходів. Для вирішення цієї проблеми за допомогою принципів ООП було розроблено архітектуру адаптерів, яка наведена на рисунку 3.7.

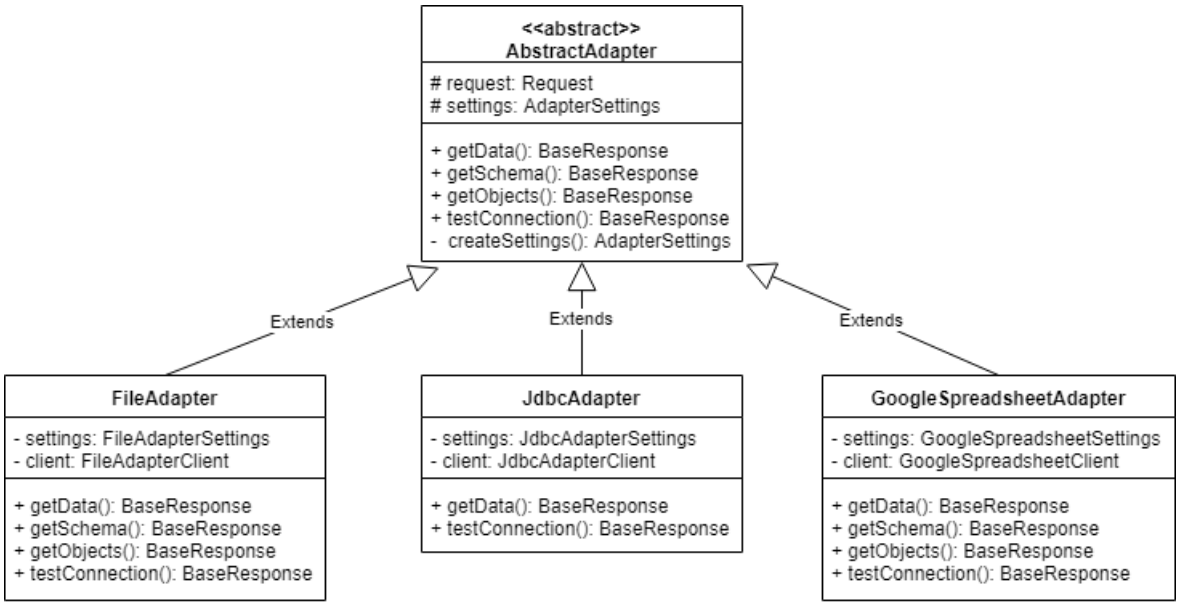

Рисунок 3.7 – Діаграма класів адаптерів

Для кожного сховища даних створюється відповідний адаптер, який інкапсулює механізми роботи з ним. Завдяки такій абстракції досягається можливість використання однакових типів даних для роботи з різними джерелами даних. Такий підхід також надає можливість легкого впровадження

адаптерів для підтримки системою нових сховищ даних та мінімізує часові ресурси, необхідні для розробки.

Абстрактний клас AbstractAdapter описує загальні механізми для роботи зі сховищами даних. Він має наступні методи:

– getData: повертає дані із сховища даних;

– getSchema: повертає тільки метадані (імена колонок та їхні типи);

– getObjects: повертає список об’єктів (наприклад, для файлового адаптеру це буде список файлів у директорії);

– testConnection: перевіряє, чи можливе з’єднання зі сховищем даних.

Усі описані вище методи не потребують обов’язкової імплементації. У разі відсутності реалізації програмою буде згенеровано виключення NotImplementedException.

Також AbstractAdapter має публічний конструктор, який приймає об’єкт запиту, що потрапляє на контролер. На базі цього запиту у конструкторі створюється об’єкт AdapterSettings, який містить необхідні налаштування для роботи із конкретним сховищем даних. Обидва об’єкти (Response та AdapterSettings) є атрибутами класу AbstractAdapter та мають захищений модифікатор доступу, що дозволяє нащадкам цього класу мати доступ до цих атрибутів. Класи об’єктів із налаштуваннями мають таку ж структуру, як і класи адаптерів (рисунок 3.3).

Для створення об’єктів класів налаштувань відповідних адаптерів був використаний паттерн проектування «Фабричний метод».

Фабричний метод (також відомий як Віртуальний конструктор) – породжуючий шаблон проектування, який надає підкласам (дочірнім класам) інтерфейс для створення екземплярів деякого класу [7]. У момент створення нащадки можуть визначити, який клас створювати. Іншими словами, даний шаблон делегує створення об'єктів спадкоємцям батьківського класу. Це дозволяє використовувати в коді програми не специфічні класи, а маніпулювати абстрактними об'єктами на більш високому рівні.

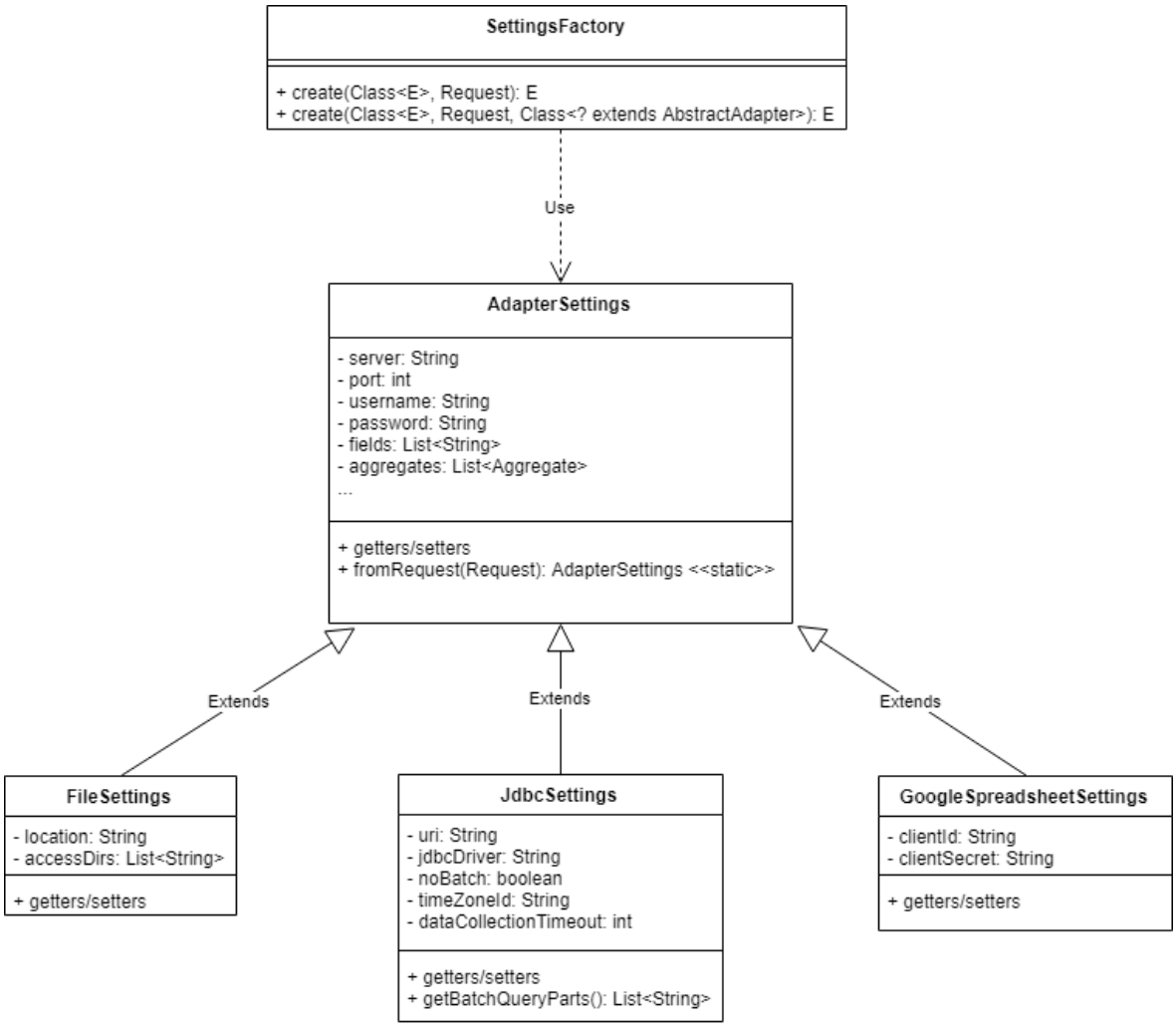

Рисунок 3.3 – Діаграма класів налаштувань адаптерів

Перевагами цього шаблону проектування є:

– дозволяє зробити код створення об'єктів більш універсальним, не прив'язуючись до конкретних класів, а оперуючи лише загальним інтерфейсом);

– дозволяє встановити зв'язок між паралельними ієрархіями класів.

Логіка для створення об’єктів налаштувань була винесена у окремий клас SettingsFactory, який на базі об’єкту запиту та типу налаштувань створює новий екземпляр. Цей клас використовує статичний метод батьківського класу AdapterSettings, який формує базовий для всіх адаптерів об’єкт налаштувань на основі вхідних даних.

На діаграмі класів адаптерів у кожній імплементації абстрактного адаптеру є атрибут «клієнт» (FileAdapterClient, JdbcAdapterClient та GoogleSpreadsheetClient). Кожен з них використовується відповідним адаптером для взаємодії із відповідним інтерфейсом прикладного програмування (API). Для класів-клієнтів не було розроблено окремої ієрархії, оскільки через різноманітність інтерфейсів доступу до сховищ даних неможливо виділити чітку абстракцію із задекларованими методами, які би вони імплементували. Таким чином, використання даного підходу не визначено програмними конструкціями, а є лише рекомендованим підходом при розробці нового адаптеру для конкретного сховища даних.

3.6 Розробка модулю перетворення даних

Після того, як дані завантажено зі сховища даних, адаптер формує об’єкт DataResponse, який містить метадані із типами та іменами колонок, та власне самі дані. Наступним етапом у процесі інтеграції є перетворення даних на основі запиту, отриманого від користувача. У системі управління сховищами даних були розроблені наступні механізми, наведені у таблиці 3.2.

Таблиця 3.2 – Функції обробки даних

Функція	Призначення
Фільтрація	Вибірка необхідних даних за критерієм пошуку
Агрегація	Угрупування даних на основі функцій-агрегацій (кількість строк, сума значень)
Сортування	Упорядкування даних на основі обраної колонки
Додавання змінних	Розширення новими даними на основі

	існуючих
Перевизначення типів даних	Перетворення типів колонок
Обмеження обсягу даних	Задання максимальної кількості строк

Усі перелічені функції підтримуються більшістю сучасних СУБД, оскільки вони надають зручний та потужний механізм маніпулюванням даними при виконанні запитів.

Всі методи перетворення даних виконуються у тому ж порядку, у якому вони зазначені у таблиці 3.2. Для цього використовується клас DataResponseCalculator, який приймає AdapterSettings як єдиний аргумент конструктору (рисунок 3.4). Цей аргумент містить усю необхідну інформацію для перетворення даних.

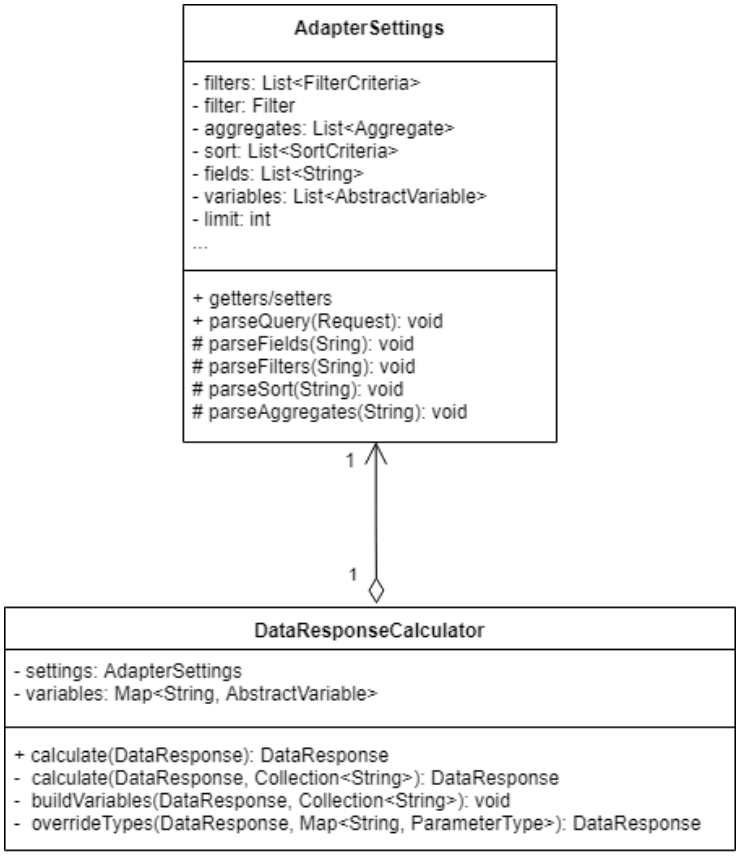

Рисунок 3.4 – Діаграма класів перетворення даних

Об’єкти AdapterSettings та DataResponseCalculator реалізують відношення агрегації «один до одного». Агрегація у ООП – відношення «частина-ціле» між двома рівноправними об'єктами, коли один об'єкт (контейнер) має посилання на інший об'єкт. Обидва об'єкти можуть існувати незалежно: якщо контейнер буде знищений, то його вміст залишається.

3.6.1 Фільтрація даних

У випадках, коли в таблиці зберігається досить велика кількість записів, виникає необхідність отримувати не всі строки, а лише ті, які відповідають певним критеріям. В SQL для цього використовується оператор «where». В системі управління сховищами даних був розроблений аналогічний механізм для фільтрації даних, якій підтримує наступні типи даних:

- текстові;
- цілочисельні;
- десяткові;
- дати.

У таблиці 3.3 вказані оператори, які підтримуються системою, та їхні значення.

Таблиця 3.3 – Оператори фільтрації

Оператор	Значення
==	Дорівнює
!=	Не дорівнює
<	Менше
<=	Менше або дорівнює
>	Більше
>=	Більше або дорівнює

is null	Запис відсутній
is not null	Запис не відсутній
in	У списку значень

На рисунку 3.5 представлена детальна діаграма класів, що характеризує структуру модулю фільтрації даних.

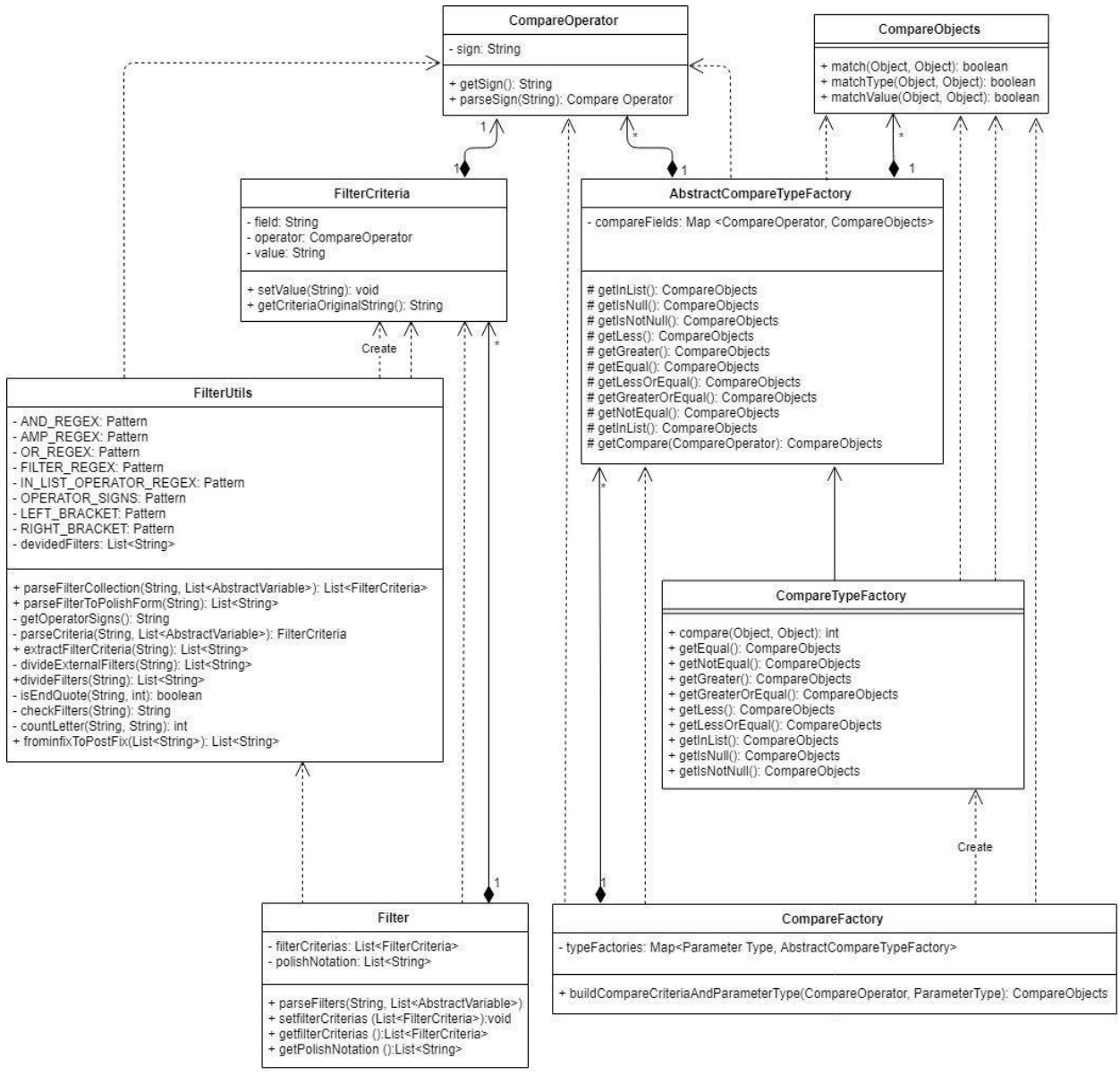

Рисунок 3.5 – Діаграма класів модулю фільтрації

Основним об’єктом у модулі фільтрації даних є FilterCriteria, який містить наступні поля:

− назва колонки, над якою буде виконуватись операція фільтрації;

− оператор, який визначається об’єктом-перерахуванням, що містить константи операторів, зазначені у таблиці 3.3;

− значення критерію фільтрування.

3.6.2 Агрегація даних

Загальною властивістю всіх сховищ даних є те, що вони містять дані з максимальним ступенем деталізації – відомості про щоденні продажі або про кожний факт продажу окремо, про обслуговування кожного клієнта тощо. Поширена думка, що таке детальне відтворення подій в досліджуваному бізнес-процесі тільки піде на користь, оскільки дані ніколи не бувають зайвими і чим більше їх буде зібрано, тим точніше виявляться результати аналізу.

Це не зовсім так. Елементарні події, з яких складається бізнес-процес, наприклад, обслуговування одного клієнта, виконання одного замовлення і так далі, які також називають атомарними (тобто неподільними), по своїй суті є випадковими величинами, схильними до впливу безліччю різних випадкових чинників. Отже, інформація про кожну окрему подію в бізнес-процесі практично не має цінності. Дійсно, на підставі інформації про продажі за один день не можна зробити висновок про всі особливості торгівлі. Таким же чином не можна виробити стратегію роботи з клієнтами на основі дослідження поведінки одного клієнта.

Іншими словами, для достовірного опису предметної області використання даних з максимальним рівнем деталізації не завжди доцільно, тому найбільший інтерес для аналізу представляють дані, узагальнені за деяким інтервалом часу, за групою клієнтів, товарів тощо. Такі узагальнені дані називаються агрегованими (іноді агрегатами), а сам процес їх обчислення –

агрегацією. В результаті агрегування велика кількість записів про кожну подію в бізнес-процесі замінюється відносно невеликою кількістю записів, що містять агреговані значення.

У даній роботі реалізовано такі функції агрегації:

− сума. Записи, що агрегуються, замінюються одним, в якому вказується сума значень;

− максимум. В результуючому записі залишається максимальне значення з усіх даних, що об’єднуються;

− мінімум. В результуючому записі залишається мінімальне значення з усіх даних, що об’єднуються;

− кількість унікальних значень. Результатом агрегації буде число унікальних значень, що з'являються у записах одного і того ж поля. Так, для поля, що містить інформацію про професії клієнта, даний спосіб агрегації покаже, скільки разів та чи інша професія з'являлася в списку;

− кількість. Результатом агрегації буде число записів, що містяться в полі;

− середнє значення. В результуючому записі залишається середнє значення записів, що об’єднуються.

На рисунку 3.6 представлена діаграма класів, які використовуються для визначення типу агрегації та назви колонки, над якою буде виконуватись функція.

Клас Aggregate містить статичний метод parse, який на основі строкового запиту формує список агрегацій. AggregateType – це клас-перерахування, який містить константи функцій агрегації.

Також клас Aggregate містить такі поля, як «поле», що визначає ім’я колонки, над якоє виконується функція агрегації, та «тип» – физначає тип агрегації, які наведено у AggregateType.

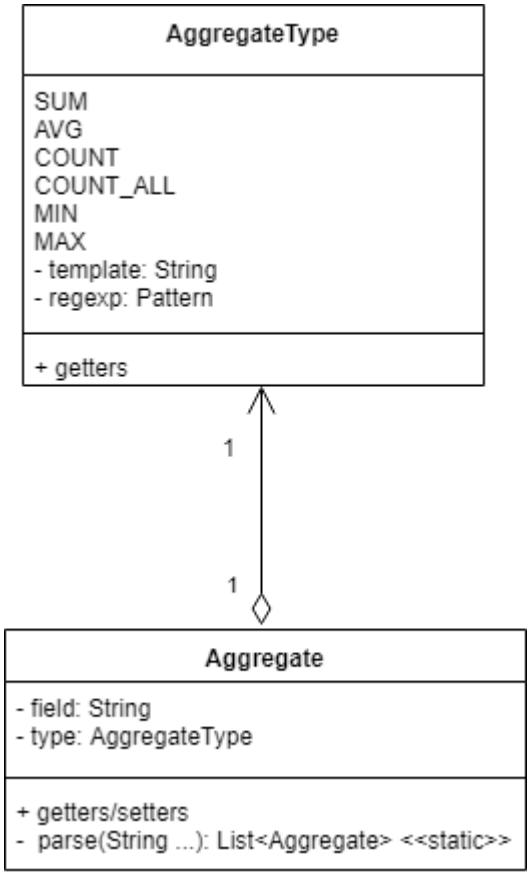

Рисунок 3.6 – Діаграма класів агрегацій

Повну структуру класів модулю агрегації даних наведено у документі ІА51.220БАК.005 Д1.

3.6.3 Сортування даних

У системі управління сховищами даних реалізовано можливість сортування у процесі пост-обробки. Цей функціонал надає можливість впорядкування результуючого набору запиту по заданому списку стовпців. Порядок, в якому рядки повертаються в результуючому наборі не гарантується, якщо не вказані критерії сортування. Аналогом у SQL є оператор ORDER BY.

На рисунку 3.7 наведено діаграму класів модулю сортування даних. Об’єкт SortCriteria інкапсулює інформацію про назву колонки, за якою відбувається сортування, та тип сортування, яких існує два види:

- за зростанням;
- за спаданням.

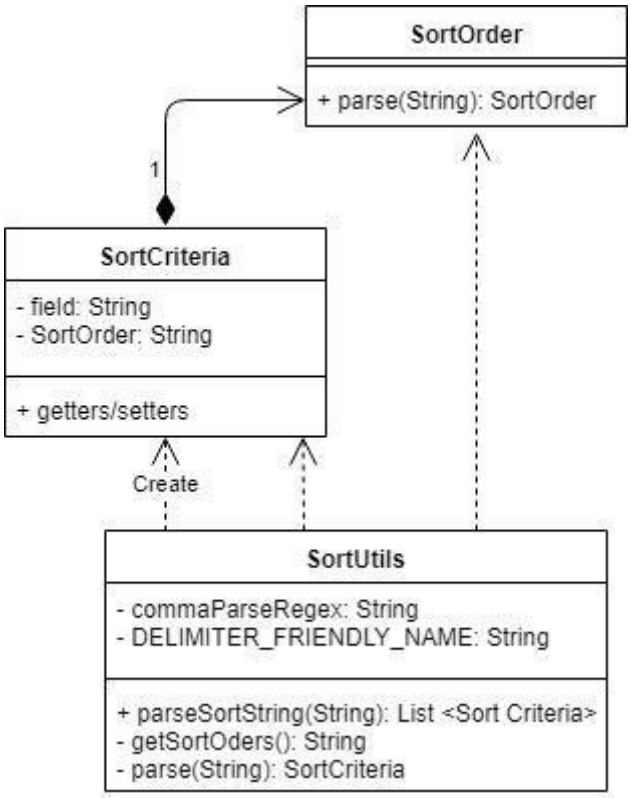

Рисунок 3.7 – Діаграма класів модулю сортування даних

Ці види представлені константами ASC та DESC відповідно у класі-перерахуванні SortOrder. Вони визначають порядок сортування значень у вказаному стовпці – за зростанням або за спаданням. Значення ASC сортує від низьких значень до високих. Значення DESC сортує від високих значень до низьких. Цей параметр не є обов’язковим, тому порядок сортування за замовчуванням – ASC. Значення NULL розглядаються як мінімально можливі значення

3.6.4 Генерація нових даних

У процесі інтеграції даних часто виникає необхідність обчислення деяких нових даних на основі існуючих, що зазвичай супроводжується створенням нових полів. Наприклад, сховище даних містить інформацію тільки про кількість і ціну проданого товару, а в цільовій таблиці потрібно мати поле «сума». Тоді в процесі перетворення необхідно обчислити суму як добуток ціни на кількість проданих одиниць товару. Таким чином, буде створено поле, що містить нову інформацію.

Генерацію нових даних розроблено на основі виразів, які представлено у таблиці 3.4:

Таблиця 3.4 – Способи генерації даних

Тип	Опис	Приклад
Константа	Використовує значення, наведені у запиті	var a = 1, var b = 'text'
Константа дати	Підтримує перелік констант, що залежать від поточної дати	var today = TODAY, var beginOfWeek = CURRENT_WEEK
Дати	Функція, що приймає від 1 до 6 параметрів (рік, місяць, день, час, хвилини та секунди)	var monthStart = date(Year, Month, 1), var newYear = date(Year, 'Jan', 1, 0, 0)
Математичні вирази	Підтримує базові математичні операції. Підтримує агрегації	var ab = [a] * [b], var x = sum(A) / [Count]

У якості констант дат було обрано наступні:

– поточний або минулий рік;
– поточна або минула чверть року;
– поточний або минулий місяць;
– поточний або минулий тиждень;
– поточний або минулий день («сьогодні» та «вчора»);
– поточний або минулий час;
– поточна або минула хвилина.

4 ОБГРУНТУВАННЯ ОБРАНИХ ТЕХНОЛОГІЙ ДЛЯ ПРОЕКТУВАННЯ СИСТЕМИ УПРАВЛІННЯ СХОВИЩАМИ ДАНИХ

4.1 Мова програмування

Java – одна з найпопулярніших мов програмування сьогодення. Розроблена у 1995 році і станом на поточний рік налічує 12 версій. Характерні особливості мови програмування Java:

- простота
- об’єктна орієнтованість;
- розподілення;
- надійність;
- безпечність;
- незалежність від архітектури комп’ютера;
- машинонезалежність;
- можливість інтерпретації;
- висока продуктивність;
- багатопоточність;
- динамічність.

Java стала технічним феноменом, що багато в чому пов’язано з її унікальною портативністю: програми Java працюють на будь-якому пристрої або операційній системі. Інтегроване середовище розробки (IDE) зробило роботу з Java набагато простішою та швидкою. Крім IDE, платформа Java має кілька інших інструментів: Maven і Ant для створення Java-програм і декомпіляторів, JConsole та VisualVM для моніторингу використання Heap. Бібліотеки з відкритим кодом полегшують використання Java у всьому світі. Apache, Google та інші організації розробили велику кількість потужних бібліотек, що полегшує та пришвидшує розробку програм. Спільнота – ще одна перевага мови і платформи Java: існує безліч активних форумів, Stack Overflow,

організації з відкритим кодом і декілька груп користувачів Java. Це дозволяє отримати допомогу тоді, коли вона дійсно потрібна.

4.2 Система збірки та керуванням життєвим циклом розробки

Maven - це інструмент для збірки Java проекту: компіляції, створення jar, створення дистрибутива програми, генерації документації. Прості проекти можна зібрати в командному рядку. Якщо збирати великі проекти з командного рядка, то команда для збірки буде дуже довгою, тому її іноді записують в bat / sh скрипт. Але такі скрипти залежать від платформи. Для того щоб позбутися від цієї залежності і спростити написання скрипта використовують інструменти для збірки проекту.

Для платформи Java існують два основні інструменти для збірки: Ant і Maven. Проте Maven має ряд переваг:

− незалежність від OS. Збірка проекту відбувається в будь-якій операційній системі. Файл проекту один і той же;

− управління залежностями. Рідко проекти пишуться без використання сторонніх бібліотек (залежностей). Ці сторонні бібліотеки часто теж використовують бібліотеки різних версій. Maven дозволяє управляти такими складними залежностями, що дає можливість вирішувати конфлікти версій і в разі потреби легко переходити на нові версії бібліотек;

− можлива збірка з командного рядка. Це часто необхідно для автоматичного складання проекту на сервері (Continuous Integration);

− зручна інтеграція з середовищами розробки. Основні середовища розробки на Java легко відкривають проекти які збираються за допомогою Maven. При цьому найчастіше проект налаштовувати не потрібно, адже він відразу готовий до подальшої розробки. Як наслідок, якщо з проектом працюють в різних середовищах розробки, то Maven є зручним способом зберігання налаштувань. Конфігураційний файл середовища розробки і для

збірки співпадає, що означає відсутність дублювання даних і відповідно помилок;

− декларативний опис проекту.

4.3 Фреймворк для розробки серверних проектів

До появи Enterprise Java Beans (EJB), розробникам Java потрібно було використовувати JavaBeans для створення веб-додатків. Хоча JavaBeans допомагали розробці компонентів користувацького інтерфейсу (UI), вони не були здатні надавати сервіси, такі як управління транзакціями та безпекою, які були необхідні для розробки надійних та безпечних корпоративних додатків. Поява EJB розглядалося як рішення цієї проблеми. Проте розробка корпоративних додатків з EJB ускладнилася, оскільки розробнику було необхідно виконувати різні завдання, такі як створення локального та віддаленого інтерфейсів, реалізація методів зворотного зв'язку з життєвим циклом, що призвело до складності надання коду для EJB.

Spring Framework з'явився як рішення для всіх цих ускладнень. Цей фреймворк використовує різні нові технології, такі як Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), та ін'єкції залежностей (DI) для розробки корпоративних додатків. Spring - це платформа з відкритим вихідним кодом, що дозволяє розробникам Java EE 7 створювати прості, надійні та масштабовані програми. Ця структура в основному зосереджена на наданні різних засобів, які допомагають керувати бізнес-об'єктами. Розробка веб-додатків набагато простіше порівняно з класичними Java-фреймворками та інтерфейсами прикладного програмування (API).

Особливості Spring Framework:

− контейнер IoC. Він відноситься до основного контейнера, який використовує шаблон DI або IoC для неявного надання посилання на об'єкт у класі під час виконання. Ця схема діє як альтернатива шаблону службового

локатора. Контейнер IoC містить ассемблерний код, який обробляє керування конфігурацією об'єктів програми. Spring Framework забезпечує два пакети, а саме org.springframework.beans і org.springframework.context, що гарантують функціональність контейнера IoC;

− система доступу до даних. Дозволяє розробникам використовувати API зберігання, наприклад JDBC і Hibernate, для зберігання даних про персистентність у базі даних. Це допомагає у вирішенні різних завдань розробника, наприклад, як взаємодіяти з підключенням до бази даних, як переконатися, що з'єднання завершено, як реалізувати управління транзакціями;

− Spring MVC framework. Дозволяє створювати веб-програми на основі архітектури MVC. Всі запити, зроблені користувачем, спочатку проходять через контролер і потім відправляються на різні сторінки JSP або сервлети. Функції форм обробки і форм перевірки форми MVC Spring Spring можуть легко інтегруватися з усіма популярними технологіями перегляду, такими як ISP, Jasper Report, FreeMarker і Velocity;

− управління транзакціями. Допомагає обробляти керування транзакціями програми, не впливаючи на їх код. Цей фреймворк надає Java Transaction API (JTA) для глобальних транзакцій, керованих сервером додатків, і локальними транзакціями, керованими за допомогою Hibernate JDBC, Java Data Objects (JDO) або інших API доступу до даних. Це дозволяє розробнику моделювати широкий спектр транзакцій на основі декларативного та програмного управління транзакціями;

− Spring Web Service. Створює кінцеві точки та визначення веб-служб на основі класів Java, але керувати ними важко в додатку. Щоб вирішити цю проблему, Веб-сервіс Spring надає багаторівневі підходи, які окремо керуються розбором XML (техніки читання та маніпулювання XML). Spring забезпечує ефективне відображення для передачі вхідного запиту XML-повідомлення об'єкту і розробнику для легкого поширення XML-повідомлення (об'єкта) між двома машинами;

− рівень абстракції JDBC. Забезпечує засоби модульного та інтеграційного тестування додатків Spring. Крім того, система Spring TestContext забезпечує особливі функції тестування інтеграції, такі як керування контекстом і кешування DI тестових приладів, а також управління транзакційними тестами з семантикою відкату за замовчуванням.

4.4 Вибір системи контролю версій

Git – різновид системи контролю версій, тобто програми для роботи з інформацією, яка постійно змінюється. Дане програмне забезпечення може зберігати безліч версій одного і того ж файлу (документу) і дозволяє повертатися до більш ранніх версій.

Git відносять до розподілених систем, тому не залежить від центрального сервера, де зберігаються файли. Git зберігає дані в локальному репозиторії — каталозі на жорсткому диску робочого комп'ютера. Для більшої стабільності і прискорення синхронізації різних версій проекту локальний репозиторій зберігають онлайн в спеціальних сервісах: Github, Gitlab, Bitbucket.

Розподілена система управління версіями, Git дотримується принципу однорангової мережі - peer to peer (рівний до рівного) на відміну від інших систем на кшталт Subversion (SVN), яка заснована на моделі client-server (клієнт-сервер).

GIT дозволяє розробникам мати безліч абсолютно незалежних гілок коду. Створення, видалення та об'єднання цих гілок відбувається без будь-яких проблем і великих витрат часу.

У Git всі операції атомарні, що означає, що будь-яка дія може бути повністю вдалою або невдалою (без будь-яких змін). Це дійсно важливо, так як в деяких системах контролю версій (на кшталт CVS), де дії не атомарні, деякі невдалі операції по всьому сховищу, можуть залишити його в нестабільному стані.

На відміну від інших VCS, таких як SVN або CVS де метадані зберігаються в прихованих папках (.cvs, .svn, і т.д.), в Git всі дані розташовані в каталогах .Git. Він використовує модель даних, яка допомагає забезпечити криптографічну цілісність всього, що є присутнім в репозиторії. Кожен раз коли файли додаються, генеруються їх контрольні суми; аналогічний процес відбувається при їх витяганні.

Ще одна важлива функція, яка присутня в Git - це його індекс. В межах індексу, розробники можуть форматувати коміти і переглядати їх до фактичного застосування.

4.5 База даних

MySQL – система управління реляційними базами даних з відкритим вихідним кодом Oracle, що базується на структурованій мові запитів (SQL). MySQL працює практично на всіх платформах, включаючи Linux, UNIX і Windows. Хоча MySQL може використовуватися в широкому діапазоні додатків, MySQL найчастіше асоціюється з веб-додатками та онлайн-публікаціями.

MySQL є важливою складовою відкритого коду підприємства, що називається LAMP. LAMP – це платформа веб-розробки, яка використовує Linux як операційну систему, Apache як веб-сервер, MySQL як реляційну систему управління базами даних і PHP як об'єктно-орієнтовану мову сценаріїв. (Іноді Perl або Python використовується замість PHP).

Спочатку задумана шведською компанією MySQL AB, MySQL була придбана компанією Sun Microsystems у 2008 році, а потім Oracle, коли вона придбала Sun в 2010 році.

Сьогодні, MySQL використовується багатьма провідними веб-сайтами у світі та незліченними корпоративними та споживчими веб-додатками, включаючи Facebook, Twitter і YouTube.

MySQL базується на моделі клієнт-сервер. Сервер MySQL доступний як окрема програма для використання в мережевому середовищі клієнт-сервер і як бібліотека, яка може бути вбудована в окремі програми. MySQL працює разом з декількома утилітами, які підтримують адміністрування баз даних MySQL. Команди надсилаються MySQLServer через клієнт MySQL, який встановлюється на комп'ютері.

База даних MySQL спочатку буал розроблена для швидкої обробки великих баз даних. Хоча MySQL, як правило, встановлюється тільки на одній машині, вона здатна розташовуватися у декількох місцях, оскільки користувачі мають доступ до неї через різні клієнтські інтерфейси MySQL. Ці інтерфейси надсилають SQL-оператори на сервер і потім відображають результати.

MySQL дозволяє зберігати дані та отримувати доступ до них через декілька механізмів зберігання, включаючи InnoDB, CSV і NDB. MySQL також здатний копіювати дані та таблиці розділів для кращої продуктивності та довговічності. Користувачам MySQL не потрібно вивчати нові команди; вони можуть отримати доступ до своїх даних за допомогою стандартних команд SQL.

MySQL написаний на C і C ++ і доступний і доступний у більш ніж 20 платформах, включаючи Mac, Windows, Linux і Unix. RDBMS підтримує великі бази даних з мільйонами записів і підтримує безліч типів даних, включаючи цілі числа 1, 2, 3, 4 і 8 байтів без знака; FLOAT; DOUBLE; CHAR; VARCHAR; BINARY; VARBINARY; TEXT; BLOB; DATE; TIME; TIMESTAMP; YEAR; SET; ENUM; і просторові типи OpenGIS. Також підтримуються типи рядків з фіксованою і змінною довжиною.

4.6 Огляд обраного середовища розробки

IntelliJ IDEA – це інтелектуальна Java IDE, яка забезпечує надійну комбінацію засобів розробки. Основні функції:

- інтелектуальна допомога в кодуванні;
- інтелектуальна навігація і пошук;
- рефакторінг;
- аналіз коду;
- підтримка веб-розробок і корпоративних розробок;
- модульне тестування і покриття кодів.

Функціональність IntelliJ IDEA постійно розширюється користувачами та третіми особами за допомогою плагінів. IntelliJ IDEA пропонує підтримку Java EE, Spring / Hibernate та інших технологічних стеків.

Інтелектуальна допомога кодування – розумний редактор, який розпізнає Java, HTML / XHTML, XML / XSL, CSS, Ruby і JavaScript, підтримує фреймворки, такі як Rails. Крім того, інтелектуальні підсвічування помилок і синтаксису в поєднанні з інтелектуальними швидкими виправленнями дозволяють ефективно створювати код і зосереджуватися на логіці проекту замість рутинної роботи.

Інтелектуальна навігація та пошук – просунута навігаційна панель проекту, яка знає про структуру коду Java, JSP, XML / XSL, HTML, Ruby та Rails, дозволяє швидко проаналізувати великі файли, надаючи зручний спосіб перегляду. Активні іконки дозволяють миттєво переходити між оголошенням символів та звичаями. Інтелектуальний пошук розпізнає елементи мови, які відображають результати на інтерактивній панелі навігації, щоб допомогти проаналізувати всі файли, в яких знайдено елемент пошуку. Крім того, функція структурного пошуку спеціально розроблена для побудови інтелектуальних шаблонів пошуку для роботи із складними файлами.

Аналіз коду – вбудований високопродуктивний динамічний аналізатор коду, з більш ніж 600 перевірками, виконує аналіз коду на льоту та виявляє всі поширені синтаксичні помилки, а також всі інші помилки, які будуть виявлені під час компіляції. IntelliJ IDEA надає аналіз для більшості підтримуваних технологій, включаючи JavaScript, XML, HTML і CSS, EJB, JSP, JSF, Struts, JavaDoc, файли властивостей, Ant сценарії та багато іншого. Аналіз статичного коду виявляє специфічні місця в продуктивності, «мертвий» код, неправильні залежності та інші проблеми. IntelliJ IDEA надає автоматичні рішення для виявлення цих помилок.

Компіляція, запуск, налагодження – IntelliJ IDEA включає підтримку декількох компіляторів Java (javac, jikes, eclipse). IntelliJ IDEA також включає в себе засновану на залежності функцію make, доступну для всіх підтримуваних компіляторів. Унікальна технологія HotSwap дозволяє змінювати та перекомпілювати частини коду під час налагодження без необхідності перезапускати весь процес.

IntelliJ IDEA надає інструмент інтелектуального покриття коду, який вимірює покриття для модульних тестів або тестів, підвищує ефективність тестування коду та гарантує ефективну перевірку коду. Крім того, IntelliJ IDEA надає гнучкий API, який дозволяє легко створювати різні плагіни, щоб розширити функціонал аналізу коду проекту та забезпечити засоби для задоволення всіх потреб проекту.

4.7 Вибір використовуваних сховищ даних

Перед тим як приступити до процесу вилучення даних, необхідно визначити, в яких саме сховищах зберігаються дані, які повинні бути інтегровані. Всі джерела можна розділити на дві групи – розташовані в корпоративних інформаційних системах і на локальних комп'ютерах окремих користувачів. З точки зору дотримання регламенту і періодичності

завантаження даних в систему доцільними є джерела з першої групи, оскільки вони також функціонують відповідно до визначеного порядку, який простіше узгодити з періодичністю завантаження даних у систему. Що стосується сховищ на локальних ПК, то зазвичай ніяких скільки-небудь строгих правил роботи з ними не встановлюється: користувачі вмикають та вимикають комп'ютери у будь-який момент часу; постійно виникають ті чи інші проблеми з мережею і т.д. Проте саме на локальних комп'ютерах часто збирається досить цінна для аналізу інформація. Тому при виборі джерел даних для завантаження необхідно враховувати наступні фактори:

– значимість даних з точки зору аналізу;
– складність отримання даних зі сховищ;
– можливе порушення цілісності та достовірності даних;
– обсяг даних у сховищі.

Як правило, доводиться шукати компроміс між цими факторами. Наприклад, дані можуть представляти безперечну цінність для аналізу, але складність їх вилучення чи некоректність структури може звести нанівець всі переваги від їх використання. В іншому випадку дані легкодоступні і не вимагають додаткової обробки при завантаженні, але при цьому практично не представляють інтересу з точки зору аналізу.

ВИСНОВКИ

У відповідності до завдання до дипломного проекту було розроблено систему управління сховищами даних.

Система надає можливість інтегрувати дані із різноструктурованих сховищ даних, зводить дані до єдиної моделі та надає уніфікований інтерфейс доступу.

Розроблена система підтримує наступні можливості:

− завантаження даних із реляційних баз даних, локальних файлів формату CSV та з веб-додатку Google Sheets;

− перетворення даних (фільтрація, агрегація, сортування, доповнення новими даними, обмеження);

− перевірка з’єднання із підтримуваними сховищами даних;

− отримання списку об’єктів зі сховищ даних.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Levy A.Y. Logic-Based Techniques in Data Integration. Logic-based Techniques in Data Integration. In: Logic Based Artificial Intelligence. Edited by J. Minker. Kluwer Publishers, 2000.

2. Manolescu I., Florescu D., Kossman D. Answering XML Queries over Heterogeneous Data Sources. Proc. Of the 27th VLDB Conference, Roma, Italy, 2001.

3. М.Р. Когаловский, МЕТОДЫ ИНТЕГРАЦИИ ДАННЫХ В ИНФОРМАЦИОННЫХ СИСТЕМАХ, Москва, 2010

4. Брюхов Д. О., Скворцов Н. А. Вилучення інформації з великих колекцій російськомовних текстових документів в середовищи Hadoop, 2014

5. Когаловский М.Р. Перспективні технології інформаційних систем. – М.: ДМК Пресс, 2003. – 288 с. 12.

6. Когаловский М.Р. Енциклопедія технологій баз даних. – М.: Фінанси та статистика, 2002. – 800 с.

7. https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html

8. Design Patterns, Джон Влисидис, Ральф Джонсон, Ричард Хелм, и Эрих Гамма, 1994

9. Batini C., Lenzerini M. A Methodology for Data Schema Integration in the EntityRelationship Model. ER 1983, pp. 413-420.

10. Гринев М.Н., Кузнецов С.Д. UQL: язык запросов к интегрированным данным в терминах UML. Программирование. – 2002. - №4

11. Калиниченко Л.А. Методы и средства интеграции неоднородных баз данных. - М.: Наука. Гл. ред. физ.-мат. литературы, 1983. - 424 с.

12. XQuery 1.0: An XML Query Language. W3C Working Draft, 15 November, 2002.

13. Веселов В., Долженков А. XML и технологии баз данных. URL: http://www.osp.ru/os/2001/02/ (дата обращения: 22.06.2011).

14. Intersoft Lab. Strategies for consolidation of disparate data and analytical applications. URL: http://www.iso.ni/joumal/articles/353.html.

ДОДАТОК А

Лістинг програми

Додаток А.1 – AbstractAggregateResult

```
package com.metricinsights.collector.plugin.aggregates;

import com.metricinsights.collector.plugin.enums.ParameterType;
import com.metricinsights.collector.plugin.types.DataField;
import com.metricinsights.collector.plugin.types.DataFieldFactory;
import com.metricinsights.collector.plugin.types.IntegerField;
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
public abstract class AbstractAggregateResult implements AggregateResult {

    protected ParameterType parameterType;
        protected int aggregationIndex;

        @Override
        public abstract void iterate(Object rowValue);

    @Override
    public abstract Object getResult();

        /**
```

```
        * Returns Long or Double representation of this value. Tries Long first, then Double.
        * @param rowValue
        * @return Long or Double depending on value of the argument
        * @throws NumberFormatException
        */
        protected Number parseNumber(Object rowValue) throws NumberFormatException {
                try {
                        if ( rowValue instanceof Short ) {
                                return Long.valueOf(((Short)rowValue));
                        } else if ( rowValue instanceof Integer ) {
                                return Long.valueOf(((Integer)rowValue));
                        } else if ( rowValue instanceof Long ) {
                                return (Long)rowValue;
                        } else if ( rowValue instanceof IntegerField ) {
                                return (Long)((IntegerField)rowValue).getObject();
                        } else if ( rowValue instanceof Number ) {
                                Number numValue = (Number)rowValue;
                                return numValue.doubleValue();
                        } else if ( rowValue instanceof DataField ) {
                                return (Double) ((DataField) rowValue).getObject();
                        } else if ( rowValue instanceof String ) {
                                return (Double) DataFieldFactory.getNumberField((String)rowValue).getObject();
                        }
                        throw new NumberFormatException();
                } catch ( Exception e ) { throw new NumberFormatException(); }
        }
```

```
        @Override
        public Integer getAggregateIndex() {
                return aggregationIndex;
        }
}
```

Додаток А.2 – Aggregate

```
package com.metricinsights.collector.plugin.aggregates;

import java.util.LinkedList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Aggregate {
   private String field;
   private AggregateType type;

   public String getField() {
      return field;
   }

   public void setField(String field) {
      this.field = field;
   }

   public AggregateType getType() {
```

```
        return type;
    }

    public void setType(AggregateType type) {
        this.type = type;
    }

    @Override
    public String toString() {
        return String.format(type.getTemplate(), this.field);
    }

    public static List<Aggregate> parse(String... aggSpecs) {
        List<Aggregate> aggregates = new LinkedList<>();
        if (aggSpecs == null || aggSpecs.length == 0)
            return aggregates;
        for (String metric : aggSpecs) {
            for (AggregateType aggregateType : AggregateType.values()) {
                Pattern pattern = aggregateType.getRegExp();
                Matcher matcher = pattern.matcher(metric);

                if (matcher.matches() && matcher.groupCount() > 0) {
                    Aggregate aggregate = new Aggregate();
                    String fieldName;
                    if (matcher.group(1).equals("*") && aggregateType ==
AggregateType.COUNT) {
                        continue;
                    } else {
                        if (aggregateType == AggregateType.COUNT_ALL) {
```

```
                    fieldName = null;
                } else {
                    fieldName = matcher.group(1);
                }
            }
            aggregate.setField(fieldName);
            aggregate.setType(aggregateType);
            aggregates.add(aggregate);
            break;
        }
    }
}

return aggregates;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (obj instanceof Aggregate) {
        Aggregate other = (Aggregate) obj;
        if (this.type != other.type) {
            return false;
        }
```

```
            return this.field == null ? other.field == null : this.field.equals(other.field);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return 31 * (type != null ? type.hashCode() : 1) + 17 * (field != null ? field.hashCode() : 1);
    }
}
```

Додаток А.3 – AggregateKey

```
package com.metricinsights.collector.plugin.aggregates;

import lombok.Getter;
import lombok.Setter;

import java.util.LinkedList;
import java.util.List;
import java.util.Map;

/**
 * An aggregate key is similar to the group by clause in SQL
 * Each dimension that we're grouping by will be added to a
 * a unique key, which looks something like:
 *
 *    dim1val1^dim2val1^dim3val1
```

```
 *
 */
public class AggregateKey {
    private static final String KEY_DELIMITER = "^";

    @Getter
    @Setter
    private List<Object> keys;

    public AggregateKey(Map<String, Object> record, List<String> dimensions) {
        keys = new LinkedList<>();
        for (String dimension : dimensions) {
            keys.add(record.get(dimension));
        }
    }

    @Override
    public int hashCode() {
        return this.getKeyString().hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof AggregateKey)) {
            return false;
        }
```

```
        AggregateKey otherKey = (AggregateKey) obj;
        return this.getKeyString().equals(otherKey.getKeyString());
    }

    private String getKeyString() {
        StringBuilder buffer = new StringBuilder();
        for (Object key : keys) {
            if (key == null)
                buffer.append("NULL").append(KEY_DELIMITER);
            else
                buffer.append(key.toString().toLowerCase()).append(KEY_DELIMITER);
        }
        return buffer.toString();
    }
}
```

Додаток А.4 – AggregateResultFactory

```
package com.metricinsights.collector.plugin.aggregates;

import com.metricinsights.collector.plugin.enums.ParameterType;

public class AggregateResultFactory {

    public AggregateResult create(AggregateType aggregateType, ParameterType type, int index) {
        switch (aggregateType) {
            case AVG:
                return new AvgResult(type, index);
```

```
            case SUM:
                return new SumResult(type, index);
            case COUNT:
                return new CountResult(index);
            case COUNT_ALL:
                return new CountAllResult();
            case MIN:
                return new MinResult(type, index);
            case MAX:
                return new MaxResult(type, index);
        }
        return null;
    }

    public AggregateResult create(AggregateType aggregateType) {
        return create(aggregateType, null, 0);
    }
}
```

Додаток А.5 – AggregateType

```
package com.metricinsights.collector.plugin.aggregates;

import lombok.Getter;

import java.util.regex.Pattern;

@Getter
public enum AggregateType {
```

```
        SUM("sum(%s)", "sum\\(\\s*(.+?)\\s*\\)"),
        AVG("avg(%s)", "avg\\(\\s*(.+?)\\s*\\)"),
        COUNT("count(%s)", "count\\(\\s*(.+?)\\s*\\)"),
        COUNT_ALL("count(*)", "count\\(\\s*(\\*)\\s*\\)"),
        MIN("min(%s)", "min\\(\\s*(.+?)\\s*\\)"),
        MAX("max(%s)", "max\\(\\s*(.+?)\\s*\\)");

        AggregateType(String template, String regexp){
            this.template = template;
            this.regExp = Pattern.compile(regexp);
        }
        private String template;
        private Pattern regExp;
}
```

Додаток А.6 – QueryResponseCalculator

```
package com.metricinsights.collector.plugin.dimensions;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.*;

import com.metricinsights.collector.plugin.pluginterface2.response.QueryResponse;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.exec.util.StringUtils;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
```

```
import com.metricinsights.collector.plugin.enums.ParameterType;
import
com.metricinsights.collector.plugin.pluginterface2.calculator.ResponseAggregator;
import com.metricinsights.collector.plugin.pluginterface2.calculator.ResponseFilter;
import com.metricinsights.collector.plugin.pluginterface2.calculator.ResponseSort;
import com.metricinsights.collector.plugin.aggregates.Aggregate;
import com.metricinsights.collector.plugin.filter.FilterCriteria;
import com.metricinsights.collector.plugin.filter.SortCriteria;
import com.metricinsights.collector.plugin.settings.PluginSettings;
import com.metricinsights.collector.plugin.types.DataField;
import com.metricinsights.collector.plugin.types.DataFieldFactory;
import com.metricinsights.collector.plugin.types.DateField;

import de.congrace.exp4j.UnknownFunctionException;
import de.congrace.exp4j.UnparsableExpressionException;

@Slf4j
public class QueryResponseCalculator {
   private final SimpleDateFormat dateFormat = new
SimpleDateFormat(DateField.MI_FORMAT);

   private Map<String, AbstractVariable> variables;
   private PluginSettings settings;

   public QueryResponseCalculator(PluginSettings settings) {
      this.settings = settings;

      variables = new HashMap<>();
```

```
    for (AbstractVariable var : settings.getVariables()) {
        variables.put(var.getAlias(), var);
    }
}

public QueryResponse calculate(QueryResponse response) throws Exception {
    log.debug("Dimensions calculation started");

    List<String> originalColumns = new ArrayList<>();
    for (QueryResponse.HeaderEntry headerEntry : response.getHeader()) {
        originalColumns.add(headerEntry.getName());
    }

    List<String> selectedDimensions = new ArrayList<>(settings.getDimensions());
    if (selectedDimensions.size() == 0 && settings.getMetrics().size() == 0) {
        log.debug("No dimensions and metrics. Use original columns");
        selectedDimensions = originalColumns;
    }

    List<String> finalColumns = new ArrayList<>(selectedDimensions);
    for (Aggregate aggregate : settings.getMetrics()) {
        finalColumns.add(aggregate.toString());
    }

    QueryResponse res = overrideTypes(response, settings.getTypeOverrides());

    // Filtering
    if (settings.getFilter().getFilterCriterias().size() > 0) {
        log.debug("Start filtering");
```

```
        // Prepare for filtering
        List<String> filterReqColumns = new ArrayList<>();
        for (FilterCriteria filterCriteria : settings.getFilter().getFilterCriterias()) {
            filterReqColumns.add(filterCriteria.getField());
        }

        log.debug("Columns, required: {}", filterReqColumns.size());

        Set<String> dimensionsForFilter = new HashSet<>(originalColumns);
        dimensionsForFilter.addAll(filterReqColumns);

        res = calculate(res, dimensionsForFilter);
        res = overrideTypes(res, settings.getTypeOverrides());

        ResponseFilter filter = new ResponseFilter();
        filter.filterResponse(res, settings.getFilter());
    }

    // Aggregation
    List<Aggregate> allAggs = new ArrayList<>(settings.getMetrics());
    // Add aggs from expressions
    for (AbstractVariable variable : settings.getVariables()) {
        if (variable instanceof ExpressionWithAggregationsVariable &&
selectedDimensions.contains(variable.getAlias())) {
            ExpressionWithAggregationsVariable tmpVariable =
(ExpressionWithAggregationsVariable) variable;
            allAggs.addAll(tmpVariable.getAggregates());
        }
    }
```

```
        if (allAggs.size() > 0) {
            log.debug("Start aggregation");
            // Prepare for aggregation
            List<String> aggReqColumns = new ArrayList<>();
            // Fields which will by added after aggregation
            List<String> aggResultColumns = new ArrayList<>();

            for (Aggregate aggregate : allAggs) {
                aggResultColumns.add(aggregate.toString());
                if (aggregate.getField() != null) {
                    aggReqColumns.add(aggregate.getField());
                }
            }

            log.debug("Columns, required: {}", aggReqColumns.size());

            List<String> selectedDimensionsWithoutAggVars = new
ArrayList<>(selectedDimensions);

            // Remove names of variables with aggs
            for (AbstractVariable variable : settings.getVariables()) {
                if (variable instanceof ExpressionWithAggregationsVariable) {
                    selectedDimensionsWithoutAggVars.remove(variable.getAlias());
                }
            }

            Set<String> dimensionsForAggregation = new
HashSet<>(selectedDimensionsWithoutAggVars);
            dimensionsForAggregation.addAll(aggReqColumns);
```

```
        // Add fields required for sort
        for (SortCriteria criteria : settings.getSort()) {
            String fieldStr = criteria.getField();
            if (!aggResultColumns.contains(fieldStr)) {
                dimensionsForAggregation.add(criteria.getField());
                selectedDimensionsWithoutAggVars.add(criteria.getField());
            }
        }

        res = calculate(res, dimensionsForAggregation);

        ResponseAggregator responseAggregator = new ResponseAggregator();
        responseAggregator.calculateAggregates(res, allAggs, new
ArrayList<>(selectedDimensionsWithoutAggVars));
    }

    // Sorting
    if (settings.getSort().size() > 0) {
        log.info("Start sorting");
        // Add fields required for sort
        // Check if the fields weren't added earlier
        if (settings.getMetrics().size() == 0) {
            log.info("Calc additional variables for sorting");
            Set<String> dimensionsForSort = new HashSet<>();
            for (SortCriteria criteria : settings.getSort()) {
                dimensionsForSort.add(criteria.getField());
            }
            for (QueryResponse.HeaderEntry headerEntry : res.getHeader()) {
```

```
                dimensionsForSort.add(headerEntry.getName());
            }
            res = calculate(res, dimensionsForSort);
        }

        ResponseSort responseSort = new ResponseSort();
        responseSort.sortResponse(res, settings.getSort());
    }

    res = calculate(res, finalColumns, true);
    res = overrideTypes(res, settings.getTypeOverrides());
    log.info("Dimensions calculation finished");

    int limit = settings.getLimit();
    if (limit > 0 && res.getData().size() > limit) {
        log.info("Size of data: " + res.getData().size() +
                ", limit: " + limit);

        List<List<Object>> data = res.getData();
        List<Object>[] dest = Arrays.copyOf(data.toArray(), limit, List[].class);
        List<List<Object>> newData = Arrays.asList(dest);
        res.setData(newData);
        log.info("Size of data after trimming: " + res.getData().size());
    }
    return res;
}
```

```
    private void buildVariables(QueryResponse response, Collection<String> requiredVariables) throws UnknownFunctionException, UnparsableExpressionException {
        // Get existing column names from header
        Set<String> existingColumns = new HashSet<>();
        for (QueryResponse.HeaderEntry headerEntry : response.getHeader()) {
            existingColumns.add(headerEntry.getName());
        }

        for (String key : variables.keySet()) {
            if (existingColumns.contains(key)) {
                continue; // Skip calculated variables
            }
            if (requiredVariables != null && !requiredVariables.contains(key)) {
                continue; // Skip unnecessary variables
            }
            AbstractVariable variable = variables.get(key);
            variable.buildVariable(response.getHeader(), settings);
        }
    }

    private List<QueryResponse.HeaderEntry> getNewHeader(QueryResponse response, List<String> dimensions) {
        log.info("Calculate new header: {}", dimensions.size());
        List<QueryResponse.HeaderEntry> res = new ArrayList<>();

        for (String dimension : dimensions) {
            QueryResponse.HeaderEntry headerEntry = response.getHeader(dimension);
            if (headerEntry != null) {
```

```
            res.add(headerEntry);
            continue;
        } else if ("now()".equalsIgnoreCase(dimension)) {
            log.info("Use now() column header");
            headerEntry = new QueryResponse.HeaderEntry(dimension,
ParameterType.DATE);
            res.add(headerEntry);
            continue;
        }
        if (!variables.containsKey(dimension)) {
            throw new IllegalArgumentException(String.format("No such column or
variable: %s", dimension));
        }
        log.info("Get header for variable: {}", dimension);
        headerEntry = variables.get(dimension).generateNewHeaderEntry();
        res.add(headerEntry);
    }
    return res;
  }

  private QueryResponse calculate(QueryResponse response, Collection<String>
newColumns) throws UnknownFunctionException, UnparsableExpressionException
{
    return calculate(response, newColumns, false);
  }

  private QueryResponse calculate(QueryResponse response, Collection<String>
newColumns, boolean orderColumns)
      throws UnknownFunctionException, UnparsableExpressionException {
```

```
        List<String> origColumns = new ArrayList<>();
        for (QueryResponse.HeaderEntry headerEntry : response.getHeader()) {
            origColumns.add(headerEntry.getName());
        }
        List<String> reqColumns = new ArrayList<>(newColumns);

        log.info("Original columns({}): {}", origColumns.size(),
StringUtils.toString(origColumns.toArray(new String[]{}), ","));
        log.info("Required columns({}): {}", reqColumns.size(),
StringUtils.toString(reqColumns.toArray(new String[]{}), ","));

        // check is calculation necessary
        if (orderColumns) {
            if (isSameOrder(origColumns, reqColumns)) {
                log.info("Request columns same as presented. Skip calculation");
                return response;
            }
        } else {
            if (isSameColumns(origColumns, reqColumns)) {
                log.info("Request columns same as presented. Skip calculation");
                return response;
            }
        }
        // Rebuild required variables
        buildVariables(response, newColumns);

        // Calculate header
        List<QueryResponse.HeaderEntry> newHeader = getNewHeader(response,
reqColumns);
```

```
List<List<Object>> newData = new ArrayList<>(response.getData().size());

log.info("Calculate new data");
// Link old columns
Map<String, Integer> headerLinks = new HashMap<>();
for (int i = 0; i < response.getHeader().size(); i++) {
   headerLinks.put(response.getHeader(i).getName(), i);
}

for (List<Object> oldDataRow : response.getData()) {
   List<Object> newDataRow = new ArrayList<>(reqColumns.size());
   for (int i = 0; i < reqColumns.size(); i++) {
      String dimension = reqColumns.get(i);
      // Search in original columns
      if (headerLinks.containsKey(dimension)) {
         int index = headerLinks.get(dimension);
         // defensive check for column not in original data
         if (index < oldDataRow.size()) {
            newDataRow.add(oldDataRow.get(index));
         } else {
            log.warn("Column index of " + index + " for column " + dimension
                  + " not found in data row with only " + oldDataRow.size() + "
columns");
            newDataRow.add(null);
         }
         continue;
      }
      // Search in variables
```

```
            if (variables.containsKey(dimension)) {
                AbstractVariable variable = variables.get(dimension);
                Object resultObj = variable.calculate(oldDataRow, newHeader.get(i));
                newDataRow.add(resultObj);
                continue;
            } else if ("now()".equalsIgnoreCase(dimension)) {
                //newDataRow.add("2014-07-31 13:58:52");
                //log.info("Use now() column dimension to calculate today's date " + timeNow);
                newDataRow.add(dateFormat.format(new Date()));
                continue;
            }
            throw new IllegalArgumentException(String.format("No such column or variable: %s", dimension));
        }
        newData.add(newDataRow);
    }

    return new QueryResponse(newHeader, newData);
  }

  private QueryResponse overrideTypes(QueryResponse response,
                                      Map<String, ParameterType> overrides) {
    log.info("Override types");
    // Link headers
    Map<String, Integer> headerLinks = new HashMap<>();
    for (int i = 0; i < response.getHeader().size(); i++) {
      headerLinks.put(response.getHeader(i).getName(), i);
    }
```

```
List<String> fieldsForOverriding = new ArrayList<>();
for (String key : overrides.keySet()) {
    if (headerLinks.keySet().contains(key)) {
        fieldsForOverriding.add(key);
    }
}
if (fieldsForOverriding.isEmpty()) {
    log.info("Nothing to override");
    return response;
}
int index;
ParameterType type;

// Override header types
for (String field : fieldsForOverriding) {
    index = headerLinks.get(field);
    type = overrides.get(field);
    QueryResponse.HeaderEntry header = response.getHeader(index);
    header.setType(type);
}

// Override data types
for (int i = 0; i < response.getData().size(); i++) {
    List<Object> row = response.getData().get(i);
    for (String field : fieldsForOverriding) {
        index = headerLinks.get(field);
        type = overrides.get(field);
        row.set(index, getAs(row.get(index), type, settings));
```

```
            }
        }
        return response;
    }

    private static Object getAs(Object obj, ParameterType type, PluginSettings settings) {
        if (obj == null) {
            return null;
        }
        try {
            switch (type) {
                case INTEGER:
                    if (obj instanceof Number) {
                        return ((Number) obj).longValue();
                    }
                    try {
                        DataField res = DataFieldFactory.getDataField(obj.toString(), type, null);
                        return ((Number) Double.parseDouble(res.getObject().toString())).longValue();
                    } catch (Exception ignored) {
                    }

                    return ((Number) Double.parseDouble(obj.toString())).longValue();

                case DECIMAL:
                    if (obj instanceof Number) {
                        return ((Number) obj).doubleValue();
```

```
            }
            try {
               DataField res = DataFieldFactory.getDataField(obj, type, null);
               return ((Number)
Double.parseDouble(res.getObject().toString())).doubleValue();
            } catch (Exception ignored) {
            }
            return Double.parseDouble(obj.toString());

         case DATE:
            if (obj instanceof Date) {
               return obj;
            }

            Date date = null;
            if (settings.getPrimaryDateFormat() != null &&
!settings.getPrimaryDateFormat().isEmpty()) {
               for (String dateFormat : settings.getPrimaryDateFormat()) {
                  try {
                     DateTimeFormatter formatter =
DateTimeFormat.forPattern(dateFormat);
                     DateTime dateTime = formatter.parseDateTime(obj.toString());
                     date = dateTime.toDate();
                     break;
                  } catch (Exception ignored) {
                  }

               }
            }
```

```
                if (date == null) {
                    date = (Date) new DateField(obj.toString()).getObject();
                }

                return date;

            case TEXT:
                if (obj instanceof Date) {
                    return new SimpleDateFormat(DateField.MI_FORMAT).format((Date) obj);
                }
                return obj.toString();

            default:
                log.error("Unknown or deprecated type {}. Return NULL", type);
                return null;
        }
    } catch (NumberFormatException | ParseException nfe) {
        log.info("Cann't cast {} to {}. Return NULL", obj, type);
        return null;
    }
}

private boolean isSameColumns(List<String> orig, List<String> req) {
    Set<String> union = new HashSet<>(orig);
    union.addAll(req);
    for (String column : union) {
        if (!orig.contains(column) || !req.contains(column)) {
```

```
                return false;
            }
        }
        return true;
    }

    private boolean isSameOrder(List<String> orig, List<String> req) {
        if (orig.size() != req.size()) {
            return false;
        }
        for (int i = 0; i < orig.size(); i++) {
            if (!orig.get(i).equals(req.get(i))) {
                return false;
            }
        }
        return true;
    }
}
```